DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Optimisation of linear algebra software implementation on a DSP architecture

MARION BELLEFIN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and
Communication Technology

Optimisation of linear algebra software implementation on a DSP architecture

MARION BELLEFIN

Master's Thesis at Ericsson AB
Ericsson Supervisor: Jakob Brundin

KTH Information and Communication Technology
Academic Supervisor: Gabriel Hjort Blindell
Examiner: Christian Schulte

TRITA-ICT-EX-2017:6

Abstract

The speed and precision of linear algebra algorithms are determining factors for the performance of wireless networks. How these algorithms are implemented is therefore critical. Even though they can be implemented with specialised hardware components, a software implementation of these functions can be preferable for its flexibility. This solution has been chosen at Ericsson to replace a hardware accelerator that currently implements a set of useful matrix functions. The aim of this project was therefore to design, implement, and optimise a software program which supports the same functions as the accelerator, and which runs on Ericsson's proprietary DSP core.

Different algorithms, data formats, and data structures have been analysed and compared based on a literature study and an analysis of the DSP's characteristics. The final program has been implemented in C for the main structure, and the actual linear algebra functions have been optimised in assembly. Several test cases have also been written and executed on a cycle-accurate simulator in order to evaluate the performance of the program in terms of speed and precision.

Even though the matrix functions are in general slower when executed on the DSP than on the hardware accelerator, the schedule of the instructions has been optimised to make the best use of the DSP's capabilities. In most cases, several versions have been written in order to adapt the data formats and the algorithms to the properties of the matrices and to the needs of the user, which was not possible on the previous hardware implementation. Bottlenecks have been identified to suggest potential future evolution in the hardware architecture and in the software interface that could further improve the performance of the program.

Referat

Hastigheten och noggrannheten i beräkningar av linjär algebra är avgörande för prestanda i trådlösa nätverk. Det är därför mycket viktigt hur man väljer att implementera dessa funktioner. Trots att de kan implementeras med dedicerad hårdvara, så kan en mjuk lösning ändå vara att föredra på grund av dess flexibilitet. Detta steg har tagits hos Ericsson för att ersätta en accelerator som utför ett antal i sammanhanget användbara matrisberäkningar. Målet med projektet har därför varit att utforma, implementera och optimera en mjukvarulösning som stöder samma funktioner som acceleratorn, och för att exekveras på Ericssons mångkärniga DSP-plattform.

Olika algoritmer, dataformat och datastrukturer har undersökts och jämförts, utifrån litteraturstudier och egenskaperna hos DSP:n. Den slutliga lösningen har implementerats i C för huvudapplikationen, och i optimerad assembler för signalbehandlingsfunktionerna. Ett antal testfall har också utformats och körts på en cykelexakt simulator, för att utvärdera exekveringstid och beräkningsnoggrannhet.

Matrisoperationerna tar generellt sett längre tid att köra på en DSP jämfört med acceleratorn, men instruktionerna har schemalagts för att på bästa sätt utnyttja DSP:ns resurser. För många funktioner har flera varianter tagits fram, för bättre anpassning till olika dataformat, matrisers egenskaper och yttre krav, vilket inte var möjligt med den existerande acceleratorn. Flaskhalsar har identifierats, och förslag till förändringar i arkitekturen och mjukvarugränssnittet har tagits fram. Införs dessa förändringar så kan prestanda ytterligare förbättras.

Contents

1 Introduction **3**
1.1 Problem statement . . . 3
1.2 Purpose . . . 4
1.3 Methodology . . . 4
1.4 Main results . . . 5
1.5 Sustainability and ethics . . . 5
1.6 Structure . . . 6

I Theoretical analysis **7**

2 Background **9**
2.1 Choice of hardware or software implementations . . . 9
2.1.1 Comparisons in the literature . . . 9
2.1.2 Why a software implementation? . . . 9
2.1.3 Existing hardware accelerator . . . 11
2.1.4 Comparison with Ericsson's proprietary DSP core . . . 13
2.2 Study of main algorithms . . . 14
2.2.1 Multiplication . . . 14
2.2.2 Division . . . 14
2.2.3 Cholesky - LDL decomposition . . . 16
2.3 Floating-point versus Fixed-point data representation . . . 16
2.3.1 Format definitions . . . 16
2.3.2 Analysis of basic arithmetic operations . . . 18
2.3.3 Matrix computations . . . 23

3 Problem Description **25**
3.1 Characteristics of the DSP . . . 25
3.1.1 Set of instructions . . . 25
3.1.2 Throughput . . . 26
3.2 Design . . . 26
3.2.1 Global program structure . . . 26
3.2.2 Matrix storage . . . 27

3.2.3 Choice of data format and algorithms 29
3.3 Analysis 31

II Implementation and Performance 35

4 Implementation 37
4.1 Information on stored matrices 37
4.2 Control of data range in fixed-point 38
4.2.1 Bit shifts in multiplication 38
4.2.2 Exponent for the ADD instruction 39
4.2.3 Control of data range 40
4.3 Load and store instructions 42
4.4 Schedule and loop unrolling 43

5 Evaluation 45
5.1 Evaluation of instructions 45
5.1.1 Accuracy 46
5.1.2 Instruction latency 50
5.2 Evaluation of a matrix algorithm 56
5.2.1 Jobs Description 56
5.2.2 Tests 57
5.2.3 Precision results 57
5.2.4 Latency results 58
5.3 Power analysis 60
5.3.1 Toggle activity 60
5.3.2 Analysis 60
5.3.3 Future work 61

6 Performance analysis 63
6.1 Identification of bottlenecks 63
6.1.1 Execution steps 63
6.1.2 MUL in floating-point 65
6.1.3 MUL in fixed-point 65
6.1.4 Other instructions with arithmetic computations 68
6.1.5 VLOAD 68
6.2 Hardware enhancements 69
6.2.1 Possible optimisations in the *shift* instruction 69
6.2.2 Load/store instructions with on-the-fly conversion 73
6.3 Suggested changes in the client interface 75
6.3.1 Revised instruction set 75
6.3.2 Matrix job templates 77

7 Conclusion 79

Bibliography **81**

Appendices **82**

A Set of matrix instructions **83**
A.1 Load - Store instructions 84
A.2 Other instructions 85

B Division algorithm **87**

Acknowledgements

I would like to thank first the supervisor of this thesis Jakob Brundin for his constant support during the few months I have worked at Ericsson. Thanks for trusting me from the beginning and all along this project.

I would also like to express my gratitude to my examiner Christian Schulte and to my academic supervisor Gabriel Hjort Blindell for their advice and their regular feedbacks on my work.

I would like to thank Pierre Rohdin for making this thesis possible and for giving me the opportunity to work on this challenging project.

A special thanks also goes to Dag Lindbo for warmly welcoming me in his team for a preliminary work prior to this thesis. Thanks for sharing your knowledge and for your kind guidance during my first months at Ericsson.

Finally I would like to thank everyone who has helped me in this project for their availability and their patience. Special thanks to Thomas Magnusson, Tomas Östlund and Ioannis Savvidis, who kindly answered my numerous questions about the different tools and without whom it would not have been possible to achieve this work.

1 Introduction

The performance of linear algebra functions is a key factor in most signal processing applications for mobile networks. Implementing these functions optimally is therefore critical. What is addressed more precisely in this project is the design and implementation of a set of functions on complex matrices in software. The target hardware is in this case Ericsson's proprietary DSP core, the latest version of which is currently under development.

1.1 Problem statement

The software program is meant to be deployed in a multi-core architecture, in which one or several Digital Signal Processors (DSPs) behave as servers. Other entities in the architecture build matrix algorithms as a sequence of matrix functions. The server is then responsible for managing the whole execution of the algorithm. These services were until now provided by a hardware accelerator that is being removed from the upcoming version of the architecture. The aim is therefore to implement the same services in software.

A simple comparison between the existing hardware accelerator and the DSP on which the software program will run reveals that the software implementation is expected to have lower performance in terms of speed if it provides the same level of precision than the accelerator. The hardware implementation has indeed been designed for that specific matrix application and contains more computational units than the DSP.

The aim of the project is therefore not to obtain the same performance as with the hardware implementation, but rather to utilise the properties of the DSP to reach the best possible performance on this new architecture. The algorithms, data structures, and data formats will be analysed and selected in this purpose.

The question that is being solved in the rest of this study is the following: How to optimise performance for linear algebra in terms of speed and precision when implemented in software on Ericsson's proprietary DSP core?

1.2 Purpose

The advantages of a software solution compared to a hardware implementation are numerous: in addition to its flexibility which can enhance the performance of the program in some specific cases, it can be easily adapted to future changes in the specifications. It can also improve the global use of resources in the multi-core architecture, since the accelerators have a very specific purpose, while the DSPs can execute a various range of applications. The number of cores dedicated to this linear algebra program can therefore vary depending on the needs of the application, and the DSPs which do not run the matrix program can be freed to execute other functions.

Furthermore, a general software program with matrix functions implemented as services can be used in many different applications, including but not restricted to Ericsson's network-related algorithms.

Throughout the optimisation of each linear algebra function and the design of the execution process (including management of data storage and control of data range), this study is also an opportunity to explore the DSP's capabilities while its latest features are being developed. The co-development of software and hardware can contribute to better performance for the former and to a design closer to the programmers' need for the latter.

1.3 Methodology

In order to make accurate design decisions, the problem has first been studied theoretically. Existing analysis in the literature have been taken into account to select the algorithms and data formats in the software. The main focus has been on algorithms' stability and speed, and the comparison of fixed- and floating-point data representation. Then the DSP architecture and characteristics have been analysed to choose accurately the data structures and to evaluate the theoretical speed of each function.

This has been followed by the actual implementation of the program. Tests have been realised with a cycle accurate simulator that emulates the upcoming version of the DSP, and the accuracy of the obtained results has been checked with Matlab software. Other simulation tools have also been used to measure factors that influence the power dissipation in different components of the architecture. The outcome includes a benchmarking of the software solution, and a comparison with the existing hardware accelerator based on quantitative results.

Finally, the limitations of the solution have been identified and some potential evolutions both in the hardware and in the software have been considered.

The different steps of the project are described in Figure 1.1.

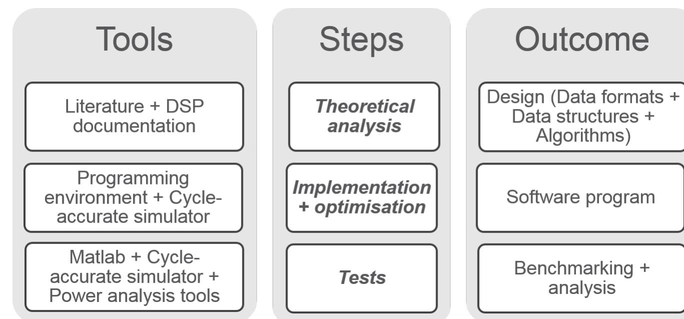

Figure 1.1. Step-by-step methodology

1.4 Main results

This study has shown that a software implementation is a viable solution for this problem. Even though it is as expected in general slower than the hardware, a schedule that makes the most of the DSP's capabilities in assembly language and a flexible choice of algorithm based on the inputs' properties can lead to better performance compared to what could be expected by a simple comparison of the hardware capabilities. The software implementation provides several versions for most functions, depending on if speed or precision is a priority for the user. Its flexibility also makes it more adaptable than the hardware to future potential changes in the specifications.

1.5 Sustainability and ethics

Sustainability aspects have also been considered, and the energy consumption of the DSP for this software program has been taken into account. This study is also part of a much wider project for the development and operation of mobile networks. Because the carbon footprint of mobile communications has kept increasing in the last decade and is expected to continue its rise [1], it is essential to take these aspects into account when considering manufacturing of devices and network operation.

While for the same functionality the energy consumption is expected to be higher for the DSP than for the hardware accelerator, the development of a flexible software solution on a general-purpose hardware has other advantages that can lead to more efficient networks. Even though they participate in the release of CO2 in the atmosphere, high-quality networks can also be the source of new solutions in energy management with the multiplication of sensors and data that require fast and dense mobile communications.

Another application to this project could be for a client DSP to directly use some of the functions of the software implementation without sending matrix jobs to a server. In this case, significant energy could be saved by not using the communication mechanism to send and receive job requests and by avoiding matrix transfers

through the shared memory. This alternative use with lower power consumption would not be possible with the hardware accelerator.

Concerning ethics, the quality and integrity of this project are guaranteed by the respect of the IEEE codes of ethics [2] by all participants. Special care has been taken to present the outcomes with objectivity, by being “honest and realistic in stating claims or estimates based on available data”.

1.6 Structure

The first chapter of this report is the present introduction. Chapter 2 describes the background of the thesis, concerning both the choice of a software implementation for this problem and numerical analysis of possible design choices. The third chapter is the continuation of the theoretical study with a deeper analysis of the problem and a description of the key aspects of the design. Then chapter 4 provides details about the actual implementation, and is followed by the evaluation in chapter 5. The last chapter aims at analysing the limitations of the design and suggesting possible evolutions both in the hardware and in the software to improve the performance of the system.

Part I

Theoretical analysis

2 Background

Before designing the software implementation for linear algebra functions, it is worth studying the current state of the art in related areas.

First, the choice of hardware or software implementations will be discussed, first in general and then for Ericsson's architecture. The aim is to understand what performance can be expected from the DSP implementation. This will be followed by a literature review concerning matrix algorithms and the choice of data format.

2.1 Choice of hardware or software implementations

2.1.1 Comparisons in the literature

Selecting the right resources for implementing signal processing algorithms is a critical choice that has been studied extensively in the literature. In order to achieve high speed requirements, fixed hardware implementation with Application-Specific Integrated Circuits (ASIC) and software implementation on Digital Signal Processors (DSP) are both popular solutions. Even though the performance is highly dependent on the DSP features (set of instructions, instruction parallelism, frequency), ASIC implementations often give better results due to the fact that they can be highly optimised for a specific purpose.

An example of comparison between ASIC and DSP implementation of a pulse-shaping FIR-filter shows that ASIC should be preferred for its speed (5 times faster at maximum clock frequencies), power consumption (more than 30 times lower), and core area [3].

Another comparison, this time for a Fast Fourier Transform (FFT) also highlights the advantages of the hardware implementation, with a comparison factor between 2 and 2.5 regarding latency and more than 10 for power efficiency [4]. However, it also highlights the lack of flexibility in ASIC, which is also mentioned by the DSP manufacturer Texas Instrument [5].

2.1.2 Why a software implementation?

In Ericsson's networks, linear algebra functions applied on matrices are currently executed on a hardware accelerator, which has been designed and optimised for a specific set of instructions. This accelerator is, however, being removed in the next

generation of the hardware platform, and an alternative software solution is now being examined. Despite their high performance, the accelerators occupy space in the multi-core chip and infer other costs even when they are not used. Replacing them by general-purpose cores could improve the global utilisation of the space and infrastructure. The design effort could therefore be more focused on Ericsson DSP which can be used for various kinds of applications. Another advantage of a software implementation is the possibility to easily update its features over time.

The targeted solution would use some DSP cores as *Matrix servers*. With a similar process as with the current hardware implementation, *Matrix clients* DSPs would send a list of matrix functions to the servers. This structure is shown in Figure 2.1. The requests are sent by the clients to a component responsible for the scheduling of the tasks. In the current configuration, the requests are forwarded to the hardware accelerators, but with the new implementation the requests will be received by other DSPs.

The number of server cores can be adjusted depending on the application, which makes this new solution much more flexible, yet completely compatible with the current solution for the client. The number of clients may vary over time as well, and those DSPs can execute different software code while waiting for the completion of their requests.

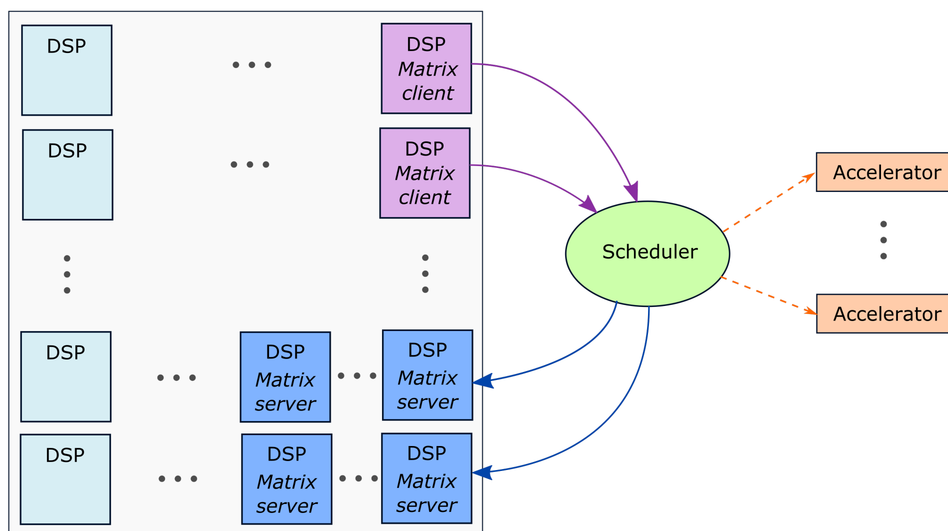

Figure 2.1. Distribution of requests

The main objective of this study is to minimise the latency in the software implementation, in order to make it a reasonable alternative to the accelerator or even a faster solution in some cases.

Another possible application of this project would be to use the code differently so that a DSP can be at the same time client and server. The scheduler would no longer be used in this case, which would lead to time and energy savings. This

application will not be examined further in this report.

2.1.3 Existing hardware accelerator

The current hardware implementation is considered as a reference in this project, in the sense that the software solution must be able to support the same specifications.

The Matrix accelerator is part of a global architecture organised around a shared memory. Communication processes enable efficient message passing between the components and transfers between the shared and the local memories.

Matrix instruction and job structures

The matrix instructions that are sent to the hardware accelerators are grouped into what is called a *matrix job*. A client which sends a job to the accelerator writes in the shared memory a *job header*, containing the number of instructions in the job, followed by the list of instructions. An *instruction* is 64-bit data structure which contains the operation code (*opcode*) of the instruction, the size of the matrices, the operands of the input and/or output matrices, and specific information regarding the instruction. The hardware accelerator is able to read, decode, and execute these instructions one after another.

A job typically begins with LOAD instructions, to load data from the shared memory to the accelerator's internal memory. Then some operations are executed internally, and finally data is written back in the shared memory with STORE instructions (see Figure 2.2). For the LOAD/STORE instructions, the input/output matrices are identified by their address in the shared memory. In all operations, each matrix is identified by an operand parameter and a size, which together refer to a unique location in the internal memory. All the internally manipulated matrices are square complex matrices whose size is a multiple of 4. LOAD and STORE instructions can operate on various matrix shapes and sizes. The accelerator also supports instructions to load and store vectors instead of matrices (VLOAD/VSTORE instructions).

JOB HEADER	
LOAD	X_1 <- addr_1
LOAD	X_2 <- addr_2
MUL	$X_0 = X_1 \cdot X_2$
SCALE_TC	$X_1 = (\lambda \cdot X_0)^*$
STORE	addr_3 <- X_1

Figure 2.2. Exemple of matrix job

Set of instructions

The following set of instructions is supported. X_0, X_1, and X_2 are complex matrices, and λ is a complex constant that can be specified in some instructions. For every complex matrix X, X^T is the transpose of X, $\overline{X}$ is the conjugate of X, and X^* is the transpose conjugate (or Hermitian transpose) of X.

- LOAD - STORE: Load/store matrix from/to the shared memory to/from the local memory
- VLOAD - VSTORE: Load/store vector from/to the shared memory to/from the local memory
- CONST: Fill a matrix with a constant λ
- SCALE: Scale a matrix by a constant ($X_0 = \lambda X_1$)
- ADD: Add a scaled matrix to another one ($X_0 = X_0 + \lambda X_1$)
- MUL: Multiply two matrices ($X_0 = X_1 X_2$)
- TMUL: Multiply a matrix with its Hermitian transpose ($X_0 = X_1 X_1^*$)
- DOT: Compute scalar product of columns of X_1^* and X_2 and write the result in the first row of X_0
- DIV: Solve linear equation ($X_0 = X_1^{-1} X_0$)
- CHOL - LDL: Compute Cholesky decomposition or LDL decomposition
- VMUL: Compute elementwise multiplication
- ABS2 - VINV - VSQRI: Other elementwise operations (write square absolute values, inverse of the real part, or inverse of square root of the real part of X_1 coefficients in X_0)
- DMAX: Regularise a matrix $X_0 = mI_d$ where m is the maximum absolute value on X_1 diagonal

For most instructions, the output of the operation can be directly transposed and/or conjugated. The full set of instructions contain for instance MUL ($X_0 = X_1 X_2$), MUL_C ($X_0 = \overline{X_1 X_2}$), MUL_T ($X_0 = (X_1 X_2)^T$), and MUL_TC ($X_0 = (X_1 X_2)^*$). The full set of instructions can be found in Appendix A.

Implementation in the Matrix Hardware Accelerator

A scratch-pad memory is included in the hardware accelerator. It can be interpreted as a $N \times N$ table containing complex coefficients written in IEEE single-precision floating-point format [6]. Depending on the size n of the matrix manipulated in the current instruction, this table is divided into up to $\lfloor N/n \rfloor^2$ blocks of size $n \times n$, and each operand in the instruction uniquely identifies one of these blocks.

All the internal computations are done with the single precision floating-point format, but the data loaded or stored from/to the shared memory can be in many different fixed-point and floating-point formats. With its floating-point units, it is able to compute up to 4 complex multiply-accumulate (*mac*) operations per cycle ($a := a + bc$) which are not always fully used. For instance, the instruction latency approaches $\frac{1}{4} \times n^3$ cycles for MUL instruction, but is closer to $\frac{1}{2} \times n^2$ for ADD. The accelerator is also able to read data easily both row-wise and column-wise in the scratch-pad memory.

2.1.4 Comparison with Ericsson's proprietary DSP core

First, the DSP and the accelerator have the same frequency, so they can be compared in terms of clock-cycle latency. Regarding computational capabilities, the DSP contains fewer floating-point units than the hardware accelerator. In the coming version of the DSP, 4 real multiply-accumulate operations can be computed per cycle when it is used optimally, which is equivalent to only 1 complex *mac* per cycle.

This next version of the DSP will also incorporate a Local Data Memory (LDM) in which the scratch-pad memory of the accelerator can be emulated. However, the access to the LDM at each cycle is limited, and no simple data structure can enable the programmer to read easily data both row-wise and column-wise (in other words to read several consecutive coefficients on the same row or on the same column in one cycle).

If the algorithms and data formats used in the software implementation are exactly the same as in the hardware implementation, the instructions' latency is expected to be higher with the software. The throughput of the most common operations is up to 4 times higher with the floating-point units of the accelerator than in the DSP. Because the computation units are not used all the time at their maximum capacity in the former, the software implementation may only be about 2 to 4 times slower only. This concerns only the execution of the instruction, opcode and operands decoding excluded. The performance of the software implementation may also be constrained by other factors such as the read/write limitations and the latency of the branching operations in the code.

Yet the main advantage of the DSP is its flexibility. While the accelerator always uses the same algorithms and data formats, the software implementation can adapt

to the different situations, for instance for the division function. Fixed-point format can also be used in some cases, which may lead to smaller latencies.

The aim of this project is now to find the best suitable algorithms, data formats and data structures, in order to obtain the best possible performance. The next sections will now review and compare different possibilities found in the literature.

2.2 Study of main algorithms

In this section and in the rest of the report, all the matrices written in capital letter are square matrices in $\mathbb{C}^{n \times n}$, unless otherwise specified. The element of A on row i and column j is written a_{ij} for $i, j \in \{0, \ldots, n-1\}$. For any complex number a and complex matrix A, $\overline{a}$ is the conjugate of a, and A^* is the conjugate transpose of A.

Most of the matrix functions described above are straightforward because they operate on one element at a time (ADD, SCALE, VMUL, etc.). The order with which the operations are executed will be selected depending the way the matrix is stored in the memory and how easy it is to read and write data in the possible configurations. The rest of this section focuses on some of the more complex instructions.

2.2.1 Multiplication

Matrix multiplication is a typical example for which the operations can be ordered in many different ways. When $C = AB$ is computed, the possible ways to organise the loops on i, j, and k to compute $c_{ij} = \sum_{i=0}^{n} a_{ik} b_{kj}$ are summarised in [7]. If for instance the inner loop is the loop on j, the data will be mostly accessed row-wise which may be helpful to achieve a better throughput.

Some algorithms to compute matrix multiplication are described in [7] and [8]. The Strassen's method which uses computation per block can for instance reduce the number of operations by a factor of approximately $7/8$, but this is only true for big matrices, and using this method would probably increase the code size and/or latency due to branching operations.

2.2.2 Division

What is meant by division here is the solving of the linear equation $AX = B$. If A is invertible, then the solution X is equal to $A^{-1}B$. [7] and [8] describe methods to solve this system in the case where B and X contain only one column and where all the matrices are real, but the methods can easily be extended.

Triangular systems

If A is either an upper triangular matrix or a lower triangular matrix, then the solution can easily be computed by the following methods. It is assumed here that

A is non-singular, which implies that all the diagonal coefficients are different from zero.

- if A is lower triangular, then the *forward substitution* algorithm gives:

$$x_{ij} = \left(b_{ij} - \sum_{k=0}^{i-1} a_{ik} x_{kj} \right) / a_{ii}$$

- if A is upper triangular, then the *back substitution* algorithm gives:

$$x_{ij} = \left(b_{ij} - \sum_{k=i+1}^{n-1} a_{ik} x_{kj} \right) / a_{ii}$$

The operations can be organised per row or per column, and it is possible to compute the result in-place by overwriting on B [7].

General systems

For general systems, a first solution would be to use **LU factorisation**, which consists in finding an unit lower triangular matrix L (with 1 on the diagonal) and an upper triangular matrix U such that $A = LU$. Such a factorisation always exists and is unique if A is invertible, and can also be found sometimes when A is singular. With this decomposition, it is easily possible to compute Y such that $LY = B$, and then X in the equation $UX = Y$.

Another popular method to solve linear systems is the **Gaussian elimination**. By using elementary row operations, the system $AX = B$ can be modified to obtain an equivalent system $A'X = B'$ in which A' is upper triangular. Then the system can easily be solved with back substitution. This method is similar to LU factorisation, but it is slightly faster when it is not needed to know the full decomposition of A (typically when only one system with A has to be solved).

A simple way to understand the algorithm is to first concatenate A and B into $[A|B] \in \mathbb{C}^{n \times 2n}$. The algorithm contains n steps, one for each row. At each step k, the current diagonal coefficient a_{kk} is used to eliminate all the coefficients below a_{kk} in this column. More precisely, for every $i > k$, the ith row row_i in $[A|B]$ is replaced by $row_i - a_{ik}/a_{kk} \times row_k$. At the end, the matrix $[A'|B']$ is obtained, with A' upper triangular (see Appendix B).

This algorithm may fail if at some point a null diagonal coefficient $a_{kk} = 0$ is encountered at step k, which may happen even when A is non-singular. Due to finite precision of data, which will be studied more explicitly in the next section, errors may also occur when a_{kk} is very closed to zero (see [8]). A solution called *partial pivoting* is to find at each step k a row row_r with $r \geq k$ such that $|a_{rk}| = \max(\{|a_{ik}| : k \leq i \leq n-1\})$, and then switch row_k and row_r (only if $r > k$) before continuing step k. Other strategies like *complete pivoting* also use column swaps for a better stability, but increase the complexity of the search for the biggest pivot.

A last method for solving the equation is **Gauss-Jordan elimination**. It is very similar to Gaussian elimination, except that instead of obtaining A' upper diagonal, the elimination technique is applied until obtaining a diagonal matrix A'' and a matrix B'' such that $A''X = B''$ is equivalent to $AX = B$. In order to obtain such a system, the step k in Gaussian elimination is modified and the operation $row_i - a_{ik}/a_{kk} \times row_j$ is applied for every $i \neq k$ (the coefficients both below and above a_{kk} are eliminated). From the last system $A''X = B''$ it is very easy to find the solution X. While Gauss-Jordan algorithm may be easier to implement, Gaussian elimination can be slightly faster and has a better stability in some cases [9].

2.2.3 Cholesky - LDL decomposition

When A is a definite positive Hermitian matrix, it is possible to find a lower triangular matrix L such that $A = LL^*$. This form is called **Cholesky decomposition** and can be used to solve linear systems. The values of the coefficients of L are given by the following explicit formulas:

$$\forall i, j \in \{0, \dots, n-1\} \text{ with } i > j,$$

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=0}^{j-1} l_{jk}\overline{l_{jk}}} \text{ and } l_{ij} = \left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}\overline{l_{jk}} \right) /ljj$$

Similarly, **LDL decomposition** is the method to find a diagonal matrix D and a unit lower diagonal matrix L (with only 1 on the diagonal) such that $A = LDL^*$. The coefficients can be computed as following:

$$\forall i, j \in \{0, \dots, n-1\} \text{ with } i > j,$$

$$d_{jj} = a_{jj} - \sum_{k=0}^{j-1} l_{jk}d_{kk}\overline{l_{jk}} \text{ and } l_{ij} = \left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}d_{kk}\overline{l_{jk}} \right) /djj$$

These decompositions can both be done in place (L overwriting A), require fewer operations than other general methods to solve linear systems, and are also more stable [8]. There are more operations in LDL decomposition than with Cholesky, but on the other hand the latter contains square root operations which can be time-consuming.

2.3 Floating-point versus Fixed-point data representation

2.3.1 Format definitions

When it comes to the digital manipulation and storage of non-integer numbers, two main approaches can be considered: fixed-point and floating-point representations.

They differ in many aspects including the supported data range, and they affect the precision and speed of the algorithm execution [10].

With the floating-point representation, specified in IEEE standard 754-2008 [6], a value is characterised by its sign, significand and exponent. While the parameters range can vary, this study will focus on the widely used 32-bit single precision format in which a value is expressed as $-1^s \times 2^{exp-127} \times 1.mantissa$. The most significant bit of the 32-bit data representation is the signed bit s, then comes 8 bits for the exponent exp interpreted as an unsigned integer, and the last 23 bits correspond to the mantissa (see Figure 2.3). This format is supported both by the Matrix hardware accelerator and by the DSP. With this data type, a very large range of data is covered (from 10^{-38} to 10^{38}) and the exponent is directly managed by the hardware during arithmetic operations.

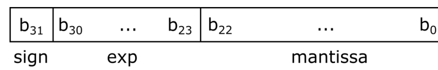

Figure 2.3. Single-precision floating-point data representation

With fixed-point representation on the contrary, the decimal point between the integer and the fractional part of a number has a fixed location, which leaves to the programmer the responsibility of dealing with all the issues related to data magnitude [10]. The data size and location of radix point are described with the $Qm.n$ notation: the m first bits represent the integer part and the n last bits the fractional part of a number. The global $(m+n)$-bit data can be interpreted as a two-complement integer that needs to be scaled by a factor 2^{-n} (see details in [11]). Figure 2.4 shows the representation of the number $-2^{m-1}b_{n+m-1} + b_{n+m-2} \dots b_n \bullet b_{n-1} \dots b_0$. The DSP provides instructions for 8-bit, 16-bit, 32-bit, and some for 40-bit data. A popular format is the $Q1.15$ (also simply called $Q15$) format for 16-bit real data between -1 and 1.

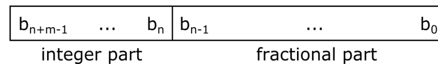

Figure 2.4. $Qm.n$ fixed-point data representation

From a hardware point of view, fixed-point arithmetic is easier to implement and often leads to a better throughput, smaller chip size and lower power consumption [12]. For a same data width, they also provide more significant bits than with the floating-point formats in which some bits are occupied by the exponent, but the latter formats deal much better with wide ranges of data [13]. The choice of formats is therefore application-dependant, and a deeper analysis is required to compare what level of precision and speed they can achieve for this specific project.

The next sections of this chapter will focus on a precision analysis, first for simple operations. It is mainly based on Wilkinson's book *Rounding Errors in Algebraic*

Processes [14] and the work of Highman in *Accuracy and Stability of Numerical Algorithms* [8].

2.3.2 Analysis of basic arithmetic operations

Notations

For any variable x, the notation $\hat{x}$ will refer to finite representation of the variable, while x will refer to its exact value. Two quantities will be useful to describe how accurately $\hat{x}$ describes x:

- the error $\epsilon_x = x - \hat{x}$
- the relative error ρ_x (defined for $\hat{x} \neq 0$ only): $x = \hat{x}(1 + \rho_x)$

The sign of both these quantities will usually be ignored.

When computing $c = op(a, b)$, where op is an arithmetic operation, three phenomena can occur.

First, the result of $op(\hat{a}, \hat{b})$ may be out of the range of data that can be written in the final format. *Overflow* happens when the absolute value of the result is too high, and *underflow* when it is too low (the latter is only a risk for floating-point format).

Another issue is that when $op(\hat{a}, \hat{b})$ is computed, some significant bits in the final result may have to be eliminated to fit in the final format This operation-related error has been studied extensively for instance in [8] and [14]. In this report, it will be described by the quantities ϵ_{op} and ρ_{op}, defined respectively by $op(\hat{a}, \hat{b}) = \hat{c} + \epsilon_{op}$ and $op(\hat{a}, \hat{b}) = \hat{c}(1 + \rho_{op})$. These quantities represent the loss of precision implied by an operation if the input data are known exactly, in other words if for every input, the finite representation in the given format is equal to the exact value of the data.

However, it is also worth studying the effects of error propagation, caused by the fact that $a \neq \hat{a}$ and $b \neq \hat{b}$. This error can be introduced as soon as the data is created or imported, or it can be caused by rounding done on previous operations in which a and b were the outputs. The consequence is that for instance $|\epsilon_c| = |\epsilon_{op} + op(\hat{a} + \epsilon_a, \hat{b} + \epsilon_b) - op(\hat{a}, \hat{b})|$ can be higher than $|\epsilon_{op}|$ [15].

For complex numbers, we assume that both the real and the imaginary parts are written in the given format. The usual norm of a complex number $x = x.re + i \cdot x.im$ with $x.re$ and $x.im$ real is $|x| = \sqrt{x.re^2 + x.im^2}$, but the infinite norm $\|x\|_\infty = \max(|x.re|, |x.im|)$ will also be useful to evaluate overflow.

Fixed-point error analysis

In this paragraph, all input and output data is represented in the $Q1.n$ format (partial results may be written in a more general $Qm.n$ format with $m \geq 1$). The

range of values can vary from -1 (included) to 1 (excluded), and the smallest non-zero absolute value is 2^{-n}. Some basic arithmetic operations can be considered, and the effects of rounding and error propagation quantified. For all the sub-mentioned arithmetic operations, we assume that for every input data, there exists an integer p such that $1 \leq p \leq n$ and for all input value x, the initial error can be bounded by $|\epsilon_x| \leq 2^{-p}$.

- **Addition - Subtraction**: these operations themselves do not involve any round-off error, $\epsilon_+ = \epsilon_- = 0$. When computing $c = a + b$, the value $\hat{a} + \hat{b}$ can be computed exactly and the result will be written in $\hat{c}$. However, an overflow may occur if the result is out of range. Furthermore, the error on the input value may propagate:

$$c = a \pm b = (\hat{a} + \epsilon_a) \pm (\hat{b} + \epsilon_b) = \hat{c} + \epsilon_c$$

  We have therefore $|\epsilon_c| \leq |\epsilon_a \pm \epsilon_b| \leq |\epsilon_a| + |\epsilon_b| \leq 2 \cdot 2^{-p} \leq 2^{-(p-1)}$

- **Multiplication**: When multiplying two $Q1.n$ numbers $c = ab$, we can easily obtain the exact result of $\hat{a}\hat{b}$ in the $Q2.2n$ format. If the result has to be written in $Q1.n$ format again, overflow may occur, but only if $\hat{a} = \hat{b} = -1$, and in this case the result can be written as $1 - 2^{-n}$ without affecting too much the accuracy of the result. When correct rounding is applied (2^{-1-n} is added to the $Q2.2n$ result before truncating it to $Q1.n$), ϵ_* can be bounded by $|\epsilon_*| \leq 2^{-n-1}$. If error propagation is also considered, then we have:

$$c = (\hat{a} + \epsilon_a)(\hat{b} + \epsilon_b) = \hat{a}\hat{b} + \hat{a}\epsilon_b + \hat{b}\epsilon_a + \epsilon_a\epsilon_b \text{ with } \hat{c} = \hat{a}\hat{b} + \epsilon_*$$

  We can deduce

$$|\epsilon_c| \leq |\epsilon_*| + |\hat{a}||\epsilon_b| + |\hat{b}||\epsilon_a| + |\epsilon_a||\epsilon_b|$$
$$|\epsilon_c| \leq 2^{-n-1} + 2 \cdot 2^{-p} + 2^{-2p} \leq 2^{2-p}$$

- **Division**: The risk of overflow is very high with fixed-point division. The result of $c = a/b$ can only be written in $Q1.n$ format if $|\hat{a}| \leq |\hat{b}|$, which is assumed for the error computation. It is also assumed that $b \neq 0$ and $\hat{b} \neq 0$. The division-related error has the same bound as the multiplication: $|\epsilon_/| \leq 2^{-n-1}$. The error on c however depends the value of b:

$$c - \frac{\hat{a}}{\hat{b}} = \frac{\hat{a} + \epsilon_a}{\hat{b} + \epsilon_b} - \frac{\hat{a}}{\hat{b}} = \frac{\epsilon_a - \frac{\hat{a}}{\hat{b}}\epsilon_b}{\hat{b} + \epsilon_b}$$

$$|\epsilon_c| \leq |\epsilon_/| + \frac{|\epsilon_a| + \left|\frac{\hat{a}}{\hat{b}}\right| |\epsilon_b|}{|\hat{b} + \epsilon_b|} \leq 2^{-n-1} + \frac{2^{1-p}}{|\hat{b} + \epsilon_b|} \text{ (with hypothesis } \left|\frac{\hat{a}}{\hat{b}}\right| \leq 1)$$

  $|b|$ should be as closed to 1 as possible in order to ensure a satisfying error bound.

	Fixed-point real	Fixed-point complex
$c = a \pm b$	$\epsilon_{\pm} = 0$ $\lvert\epsilon_c\rvert \leq 2^{1-p}$ $\lvert c\rvert \leq 2$	$\epsilon_{\pm} = 0$ $\lvert\epsilon_c\rvert \leq 2^{1-p}$ $\lVert c\rVert_\infty \leq 2$
$c = ab$	$\lvert\epsilon_*\rvert \leq 2^{-1-n}$ $\lvert\epsilon_c\rvert \leq 2^{2-p}$ $\lvert c\rvert \leq 1$	$\lvert\epsilon_*\rvert \leq 2^{-1-n}$ $\lvert\epsilon_c\rvert \leq 2^{2-p}$ $\lVert c\rVert_\infty \leq 2$
$c = \sum_{i=1}^{2^k} a_i b_i$	$\lvert\epsilon_{acc}\rvert \leq 2^{-1-n}$ $\lvert\epsilon_c\rvert \leq 2^{k+2-p}$ $\lvert c\rvert \leq 2^k$	$\lvert\epsilon_{acc}\rvert \leq 2^{-1-n}$ $\lvert\epsilon_c\rvert \leq 2^{k+2-p}$ $\lVert c\rVert_\infty \leq 2^{k+1}$

Table 2.1. Effects of fixed-point computations on precision and bounds

- **Product accumulation**: A common operation in linear algebra is the sum $c = \sum_{i=1}^{N} a_i b_i$. In order to increase the precision, it is possible to store all the partial results in an accumulator that can contain $Q2.2n$ data, and convert the result back to $Q1.n$ format only at the end of the computation [14]. In this case, the error caused by the final rounding is only $|\epsilon_{acc}| \leq 2^{-1-n}$. If error propagation is taken into account, then we have:

$$|\epsilon_c| \leq 2^{-1-n} + \sum_{i=1}^{N}(2^{1-p} + 2^{-2p}) \leq 2^{\lceil \log_2(N) \rceil + 2 - p}$$

  Overflow is likely to occur, especially if N is big, but it can be avoided by first scaling all the input values with an accurate factor. Another solution is to store the partial results in a larger accumulator (in $Qm.n$ format with $m \geq \lceil \log_2(N) \rceil$), and to scale the final result so that it can be written in $Q1.n$ format. In each case, the scaling may affect the precision.

All these results can easily be extended for complex arithmetic, with some adaptation when evaluation the risks of errors and overflow. The results are summarised in Table 2.1. The inputs and outputs are written in $Q1.n$ format but partial results are stored as $2+2n$-bit data, including partial products in the complex multiplication. For all complex input x, $\|x\|_\infty \leq 1$, which implies $|x| \leq \sqrt{2}$.

Floating-point error analysis

In this paragraph, only IEEE single-precision format is considered, in which the number of significant digits is $t = 23 + 1 = 24$. The range of values in this format is assumed to be large enough to avoid both overflows, and underflows will be ignored

as well. For this analysis, the relative error is more relevant to consider so now we assume that there exists $p \geq 1$ such that for every input $x = \hat{x}(1 + \rho_x)$, we have $|\rho_x| \leq 2^{-p}$.

An important result proven in [8] is that for every simple arithmetic operation $(+, -, \times, /)$ applied on real number, $|\rho_{op}| \leq 2^{-t}$. The effect of error propagation will now be studied in more details:

- **Addition - Subtraction**: When computing $c = a \pm b$ (assume result non-zero in finite representation, and $|a| \geq |b|$), we have:

$$c = \hat{a}(1 + \rho_a) \pm \hat{b}(1 + \rho_b) = (\hat{a} \pm \hat{b}) \left(1 + \frac{\hat{a}\rho_b + \hat{b}\rho_a}{\hat{a} \pm \hat{b}}\right)$$

  From which we can deduce:

$$1 + \rho_c = (1 + \rho_\pm) \left(1 + \frac{\rho_b + \frac{\hat{b}}{\hat{a}}\rho_a}{1 \pm \frac{\hat{b}}{\hat{a}}}\right)$$

$$|\rho_c| \leq \rho_\pm + \left|\frac{1}{1 \pm \frac{\hat{b}}{\hat{a}}}\right| (\rho_b + \rho_a)(1 + \rho_\pm) \leq 2^{-t} + \left|\frac{1}{1 \pm \frac{\hat{b}}{\hat{a}}}\right| (2^{1-p} + 2^{1-p-t})$$

  If $1 \pm \frac{\hat{b}}{\hat{a}} \geq 1$ an upper bound can be $|\rho_c| \leq 2^{2-p}$. However, the closer to zero the result of the operation gets, the bigger the relative error becomes. This is called a *cancellation effect*, and is due to the fact that when two close quantities are subtracted, almost all the significant digits are eliminated, and the result cannot be written with sufficient precision any more.

- **Multiplication**: The operation $c = ab$ gives the following:

$$c = \hat{a}\hat{b}(1 + \rho_a)(1 + \rho_b) = \hat{c}(1 + \rho_*)(1 + \rho_a)(1 + \rho_b)$$

$$|\rho_c| = |\rho_* + (\rho_a + \rho_b + \rho_a\rho_b)(1 + \rho_*)| \leq 2^{-t} + 2^{1-p} + 2^{1-p-t} + 2^{-2p} + 2^{-2p-t} \leq 2^{2-p}$$

- **Division**: When computing $c = a/b$, we have:

$$c = \frac{\hat{a}(1 + \rho_a)}{\hat{b}(1 + \rho_b)} = \hat{c}(1 + \rho_/)\frac{(1 + \rho_a)}{(1 + \rho_b)}$$

$$|\rho_c| = \frac{(1 + \rho_/)(1 + \rho_a) - (1 + \rho_b)}{1 + \rho_b} \leq 2 \cdot (2^{-t} + 2^{1-p} + 2^{-p-t}) \leq 2^{3-p}$$

  If $p \geq 3$, the bound can be slightly better: $|\rho_c| \leq 2^{2-p}$.

	Floating-point real	Floating-point complex
$c = a \pm b$	$\lvert\rho_\pm\rvert \leq 2^{-t}$	$\lvert\rho_\pm\rvert \leq 2^{-t}$
$c = ab$	$\lvert\rho_*\rvert \leq 2^{-t}$ $\lvert\rho_c\rvert \leq 2^{2-p}$	$\lvert\rho_*\rvert \leq 2^{2-t}$ $\lvert\rho_c\rvert \leq \max(2^{3-t}, 2^{2-p})$
$c = a/b$	$\lvert\rho_{acc}\rvert \leq 2^{-t}$ $\lvert\rho_c\rvert \leq 2^{3-p}$	$\lvert\rho_{acc}\rvert \leq 2^{3-t}$ $\lvert\rho_c\rvert \leq \max(2^{5-t}, 2^{3-p})$

Table 2.2. Effects of floating-point computations on precision

- **Product accumulation**: When $c = \sum_{i=1}^{N} a_i b_i$ is computed, a result from [8] gives:

$$\left|\hat{c} - \sum_{i=1}^{N} \hat{a_i}\hat{b_i}\right| \leq \frac{N2^{-t}}{1 - N2^{-t}} \sum_{i=1}^{N} |\hat{a_i}||\hat{b_i}| \text{ so } |\rho_{acc}| \leq \frac{N2^{-t}}{1 - N2^{-t}} \frac{\sum_{i=1}^{N} |\hat{a_i}\hat{b_i}|}{\left|\sum_{i=1}^{N} \hat{a_i}\hat{b_i}\right|}$$

  If for every $i \in \{1 \ldots N\}, \hat{a_i}\hat{b_i} > 0$, and if $\lceil \log_2(N) \rceil < t - 1$ (last hypothesis very likely to be true), we can deduce $|\rho_{acc}| \leq 2^{1+\lceil \log_2(N) \rceil - t}$. Otherwise, the effects of cancellation cannot guarantee an easy bound, both on ρ_{acc} and ρ_c

In the case of complex arithmetic, [8] gives some bounds on the operation error:

$$|\rho_\pm| \leq 2^{-t},\ |\rho_*| \leq \sqrt{2}\frac{2^{1-t}}{1 - 2^{1-t}} \leq 2^{2-t},\ |\rho_/| \leq \sqrt{2}\frac{2^{2-t}}{1 - 2^{2-t}} \leq 2^{3-t}$$

The error on the result depend on the value of p and t. The results are summarised in Table 2.2.

Summary

In order to compare the results for fixed- and floating-point computations, the relation between the absolute and relative error $|\rho_x|\,|\hat{x}| = |\epsilon_x|$ can be used. If $|\hat{x}|$ remains close to 1 (that will be true if data is correctly scaled in $Q1.n$ format), both metrics are comparable. For the same number of significant bits ($n = t$), then the two data formats provide similar precision. In the worst case, the number of correct significant bits (in an intuitive sense, see [8] for proposals of definition) decreases approximately by 1 for every addition, 2 for a real multiplication, and decreases logarithmically with N when N products are accumulated. However, with its automatic scaling, floating-point computations should be preferred when possible, especially when the range of variables varies a lot. The only advantage of fixed-point concerning precision is that partial results can easily be stored in formats which provide more significant bits before rounding the final results (see for

instance product accumulation). However, this will not increase the accuracy of the result in the worst case if there were already errors on the input data.

With fixed-point computations, the scaling of data needs to be constantly managed by the programmer. The aim is first to avoid overflows (which implies right-shifts of data) and secondly to maximise the number of significant bits and therefore the precision by keeping the values close to 1 (which implies left-shifts of data). Furthermore, dividing in fixed-point may not be possible with reasonable effort for some algorithms, so data would need to be converted in floating-point before executing these functions. These extra operations can be time-consuming, and question the use of fixed-point for faster computations.

2.3.3 Matrix computations

If the data is stored using $Q1.n$ format, a common exponent for a whole matrix will typically be used to scale the data. If all coefficients are approximately in the same range (in other words if $(\max(\{|a_{ij}| : i,j \in \{1\dots n\}\})/\min(\{|a_{ij}| : i,j \in \{1\dots n\}\})) \leq 2^k$ where k is no greater than a few units), then it is possible to find an appropriate common exponent which maintain all coefficients close to 1 in norm. It is only if this condition is verified that the errors caused by fixed-point computations will remain reasonable compared to errors created by floating-point computations.

Concerning the algorithms which involve divisions, a prior analysis must be done before using fixed-point. In some cases such as the Gauss-Jordan elimination, it will be necessary to do the computations in floating-point, because no hypothesis can be made on the inputs of the divisions. The possibility to implement this algorithm in fixed-point has for instance been studied in [12], but it relies on a complex dynamic analysis of the data range, which is not applicable here.

The questions that need to be solved in this study are the following: is it possible to find a scaling scheme in fixed-point which guarantees the absence of overflow without losing too much precision? Is the gain in time provided by fixed-point computation greater that the latency added by the scaling and conversion operations?

3 Problem Description

In order to obtain an optimal software implementation of the matrix functions, the characteristics of the DSP on which the program will run need to be taken into account. The design choices also rely on a performance analysis.

3.1 Characteristics of the DSP

3.1.1 Set of instructions

The instruction set of the DSP contains arithmetic operations for fixed-point and floating-point formats, read/write instructions to move data from/to the Local Data Memory to/from registers, among other operations. The DSP also implements a load/store architecture between the LDM and the shared memory. Up to 4 arithmetic operations can be executed each clock cycle. The combination of instructions executed every cycle is also constrained by the maximum number of moves and the size of the instructions in memory.

The following operations are supported for IEEE single-precision floating-point real format: addition, subtraction, multiplication, multiply-accumulate operation ($c = c + ab$), multiply-subtract operation ($c = c - ab$), division, square root, inverse of square root. These instructions have a latency of a few cycles, but all of them except the last three can be pipelined to reach the maximum of 4 instructions per cycle.

Concerning fixed-point computation, it is possible to add, subtract, multiply, multiply-accumulate, or multiply-subtract real but also complex data in $Q1.15$ format. For operations which involve multiplications, the result can also be stored in 32 or 40 bits, using $Q2.30$, $Q1.31$, $Q10.30$, or $Q9.31$ formats. Even though it has been proven that this will not increase precision in the worst case when the input data contain errors, this may increase the accuracy of the result in the general case when these larger accumulators are used to store the partial result of a sum of products. The result of these operations can also easily be right-shifted by a defined number of bits in order to avoid overflows. Division and square-root operations are also supported.

Real data can be converted from one format to another with scaling by an exponent given by the programmer. Before converting data to fixed-point, the maximum exponent of the floating-point data must be computed to avoid overflows.

	Floating-point	Fixed-point
Addition - Subtraction	2	4
Multiplication	1	4
Multiplication-accumulate/subtract	1	4

Table 3.1. Maximum number of operations per cycle for complex data

3.1.2 Throughput

The throughputs of the most used instructions are summarised in Table 3.1. They concern arithmetic operations on complex data. When complex operations are not directly available, they can be computed by combining real operations, for instance 2 real multiplications, 1 multiply-accumulate, and 1 multiply-subtract for a complex multiplication.

3.2 Design

3.2.1 Global program structure

The DSPs which receive matrix jobs must be able to fetch, decode, and execute each matrix instruction. The first steps of this process will be written in C language, but the actual matrix computations must be written in assembly language to achieve better performance. The global structure of the program is described in Figure 3.1.

Automatically compiled code does not completely make use of the DSP instruction parallelism, which considerably reduces the speed of execution. A first evaluation of program execution of simple matrix functions has shown that C code can be up to 10 times slower than optimised assembly code. However, all the transitional steps between the actual matrix computations will be written in C, because it is essential that these parts are easily readable and that they can easily be modified if the specifications change in the future. The possibility in C to name variables and data types, as well as the possibility to write concise and easily maintainable code are the key reasons to choose this language over assembly for these steps of the matrix program.

Several versions of every matrix function will be written, in order to adapt to the situation or to the client's requirements.

First, all functions will have a floating-point version, in case the client gives priority to precision over speed in the computations. When overflow can be avoided, a fixed-point implementation of the function will also be written, which will be faster but potentially less accurate than the floating-point versions in some cases.

The shape of a matrix can also influence the selection of the best algorithm. A useful example concerns matrix division: when computing $A^{-1}B$, the algorithm

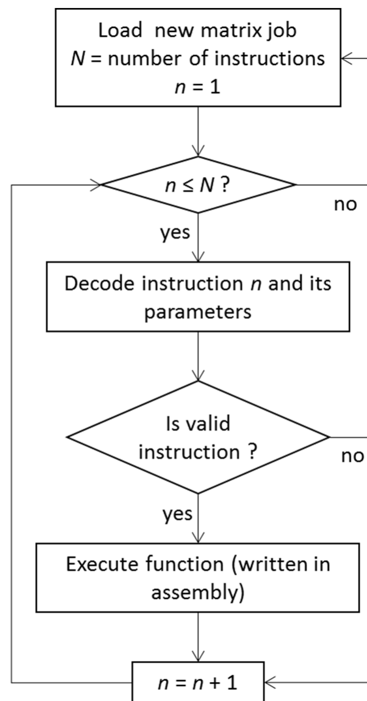

Figure 3.1. Structure of the code for a Matrix server DSP

may be adapted if A is a triangular matrix obtained for instance with a Cholesky decomposition. This way, the software implementation may not be faster than the existing hardware implementation in general but it can surpass it for some special cases.

The flexibility of the software implementation also relies on the fact that it can easily evolve. If some new use cases are identified by the programmers, a specific version of the functions can be written to obtain the best possible performance in those cases. One example could be the use of matrices in $\mathbb{R}^{n\times n}$ instead of $\mathbb{C}^{n\times n}$, which can lead to higher speed.

Having many different versions of each matrix function can increase the performance, but also the size of the code. In addition to be an issue due memory size constraint, this can also make the code less maintainable over time. It is therefore important to carefully identify the cases for which a specific implementation will actually be used and increase the speed significantly.

3.2.2 Matrix storage

The local data memory of the DSP will contain a global data structure to store the matrices, that will be called *smx_mem*, for "Soft matrix memory". It is assumed that the DSP must be able to store as many matrix coefficients as the hardware

accelerator, that is to say N^2 coefficients. Three possible way to organise *smx_mem* are described in Figure 3.2.

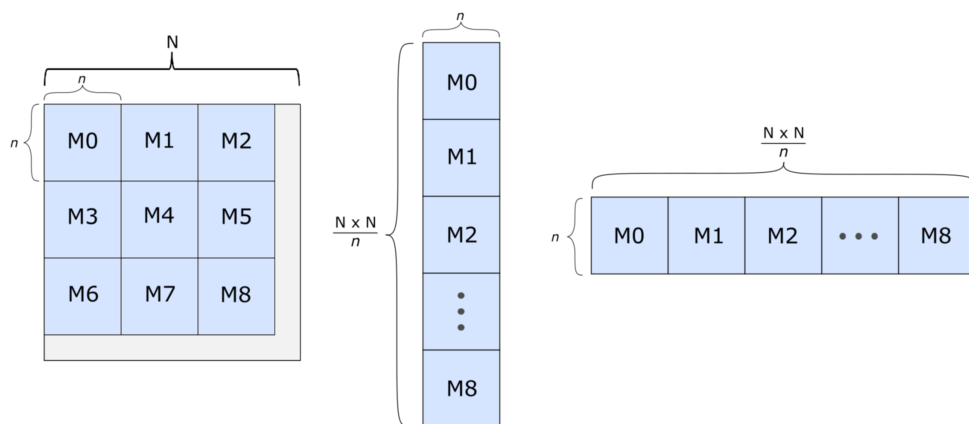

Figure 3.2. Three possible structures for *smx_mem*

- First solution (on the left): the structure is the same as in the hardware accelerator. The main data structure is a table with a N rows and N columns. The rows are stored consecutively in memory. When matrices of size n are considered, the table is divided into blocks of size $n \times n$, as shown in the Figure.
- Second solution (in the middle): each matrix is stored so that its rows are stored consecutively in memory. The matrices are stored one after another in the structure.
- Third solution (on the right): when matrices of size n are considered, *smx_mem* is a table with n rows stored consecutively, and with $\lfloor \frac{N \times N}{n} \rfloor$ columns. The division in blocks of size $n \times n$ is shown in the Figure.

A first way to compare the structures is to consider the implementation of the LOAD, STORE, VLOAD, and VSTORE functions. In the first two instructions, a full matrix of n_{rows} rows and n_{cols} columns is transferred to/from a block Mk in *smx_mem*. With VLOAD or VSTORE, the data is loaded or stored into a row that crosses successively $M0, M1, M2, M3$, etc. (see Appendix A). Here only the simple case is considered, where there is no need to transpose, conjugate or convert data between the shared memory and the DSP memory.

Because there is a constant cost to initiate a data transfer between the DSP and the shared memory, the DSP should avoid sending multiple load/store requests for small sets of data. Typically for the LOAD instruction, it should be avoided to send one load request per matrix row. If the whole matrix cannot be transferred with one request, like in configurations 1 and 3 of *smx_mem* (because the rows of the

destination matrix are not stored consecutively), the data should be for instance loaded in an intermediate buffer at first, and then copied into *smx_mem* row by row.

As a result, the only configuration in which the LOAD and STORE instructions can be implemented easily with only one transfer request is the second configuration, and only when $n_{cols} = n$ (otherwise if $n_{cols} < n$, the destination rows are not consecutive in memory). However, VLOAD and VSTORE cannot be implemented easily with this configuration, as they could be with the third configuration. With configuration 1, it is also possible to load/store vectors directly in *smx_mem* with a limited number of request (for instance with the block division of Figure 3.2, one request to fill the row in $M0, M1$, and $M2$, one request for $M3, M4$, and $M5$, etc.)

Apart from the previous considerations, the main advantage of the first solution is that it can easily support different sizes of matrices in the same job, as it is done in the hardware accelerator. This could be done but would be much more complex with the two other solutions. Even though the size n of the matrices is a constant in the most common use cases, different sizes are supposed to be supported in the specifications, this is why the first configuration is finally chosen to store the matrices. If the possibility to handle different sizes in the same job is removed from the specifications in the future, solutions 2 and 3 can be considered.

There is another issue with this first structure which concerns read conflicts. The DSP cannot read in the same cycle data which have the same alignment in memory. It cannot for instance read simultaneously data from the addresses $addr_a$ and $addr_b$ if there exists an integer k such that $addr_b - addr_a = kN$. This implies that two consecutive coefficients in a column cannot be read in the same cycle.

In order to solve this issue, it is possible to insert a fixed offset of a few bytes between each row of the $N \times N$ table. In this way, column-consecutive coefficients will not have the same alignment in the Local Data Memory.

This characteristic is illustrated in Figure 3.3, where *smx_mem* is displayed with and without an offset between the rows. Two different coloured coefficients cannot be read in the same cycle. With the offset, *smx_mem* occupies more space in memory but enables new interesting combinations of read instructions in one cycle.

3.2.3 Choice of data format and algorithms

In addition to a floating-point version, all the algorithms that do not involve any division operation will also be written in fixed-point format. Every time addition, subtraction, and complex multiplication are involved, the data will be right shifted in order to avoid overflows: only one right shift when a simple multiplication is done for each coefficient (ex: VMUL), and $1 + \lceil \log_2(n) \rceil$ shifts when up to n products are accumulated (ex: MUL). A special care has to be taken for the ADD instructions, for which we must ensure that the matrices which are added have the same exponent.

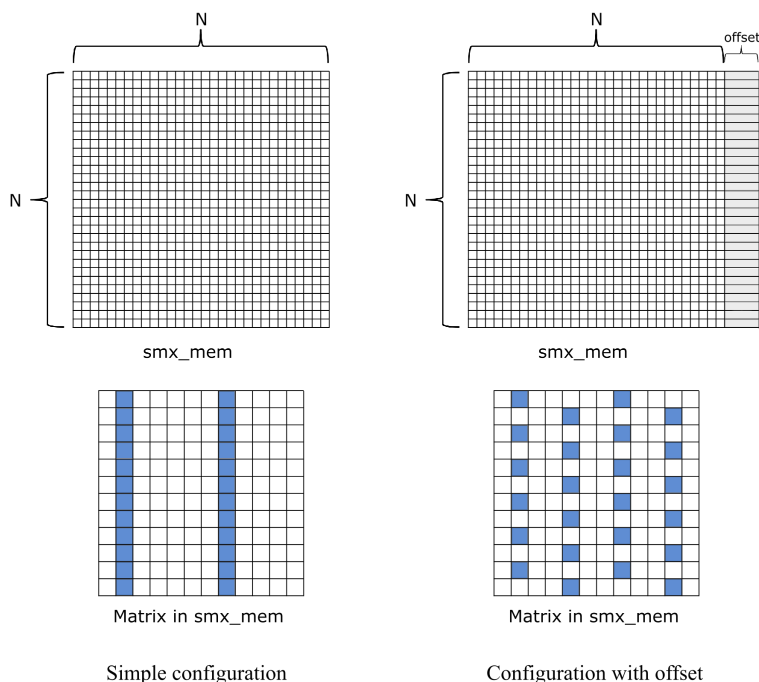

Figure 3.3. *smx_mem* configurations and consequences: only one coloured coefficient can be read each cycle by the DSP

Even though it contains divisions, the algorithm for the Cholesky decomposition can be written in fixed-point $Q1.15$ format without risking overflow if the input matrix A is Hermitian definite positive.

Proof: let A be a positive Hermitian definite matrix in $\mathbb{C}^{n\times n}$ written in $Q1.15$ format. There exists a lower triangular matrix L such that $A = LL^*$, and we have for every $i, j \in \{0, \ldots, n-1\}, i > j$,

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=0}^{j-1} l_{jk}\overline{l_{jk}}} \text{ and } l_{ij} = \left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}\overline{l_{jk}}\right)/ljj$$

This implies that for every $j \in \{0, \ldots, n-1\}$, $\|a_{jj}\|_\infty = |a_{jj}.re| = \sum_{k=0}^{j} |l_{jk}|^2 \leq 1$ (A is Hermitian so a_{jj} is real).

When the coefficients l_{ij} are computed ($i \geq j$), the products $l_{ik}\overline{l_{jk}}$ are subtracted one by one, and after step p we have ($0 \leq p \leq j-1$):

$$l_{ij}^{(p)} = a_{ij} - \sum_{k=0}^{p} l_{ik}\overline{l_{jk}} = l_{jj}l_{ij} + \sum_{k=p+1}^{j-1} l_{ik}\overline{l_{jk}} = \sum_{k=p+1}^{j} l_{ik}\overline{l_{jk}} \text{ (because } l_{jj} \text{ is real)}$$

Cauchy-Schwarz inequality gives:

$$\left|l_{ij}^{(p)}\right|^2 = \left|\sum_{k=p+1}^{j} l_{ik}\overline{l_{jk}}\right|^2 \leq \sum_{k=p+1}^{j} |l_{ik}|^2 \sum_{k=p+1}^{j} |l_{jk}|^2 \leq 1 \cdot 1$$

And $\left\|l_{ij}^{(p)}\right\|_\infty \leq \left|l_{ij}^{(p)}\right| \leq 1$. The last square-root or division operation will result in the final result of l_{ij} which also have an infinite norm lower or equal to 1 thanks to the first inequality. $\square$

Because numbers have a finite representation, the computations may in practice generate small overflows, but the results can be rounded to representable numbers with low impact on the precision.

With Gaussian elimination, the partial pivoting ensures that the division operations a_{ik}/a_{kk} with $i > k$ always produces a result bounded by 1 because a_{kk} is chosen as the highest possible pivot in the column k. However, when the coefficients are eliminated by the operations $a_{ij} \leftarrow a_{ij} - a_{ik}/a_{kk} \times a_{kj}$, nothing guarantees that the successive accumulations will not generate overflows in fixed-point. As a result, all the coefficients will need to be right-shifted before the beginning of the algorithm.

During back-substitution or Gauss-Jordan elimination algorithm, the divisions are likely to generate overflow, that is why these algorithms will only be written in floating-point.

Three different versions of the division algorithm can therefore be written in the general case: Gaussian elimination in fixed-point followed by data conversion and back-substitution in floating-point, Gaussian elimination and back-substitution both in floating-point, and Gauss-Jordan elimination in floating-point. When the input A in the computation of $A^{-1}B$ is a triangular matrix, only forward- or backward-substitution needs to be executed. Another special case is when B is only a column (all other coefficients equal to 0), which will considerably reduce the number of operations.

3.3 Analysis

Given the available set of instructions in fixed-point and floating-point in the DSP, we can deduce the total number of required DSP instructions to compute the result of matrix functions. The results for the main functions are shown in Table 3.2. Matrices in $\mathbb{C}^{n \times n}$ are considered. All the arithmetic operations involved in the computations have a low latency and can easily run in parallel (several operations per cycle).

This shows that computations in fixed-point are expected to be 2 to 4 times faster than floating-point computations. It must however be noted that some operations will need to be added in fixed-point computations in order to manage the data range. These additional operations will hopefully not affect the complexity of the matrix functions in a significant way.

	Fixed-point	Floating-point
MUL	n^3	$4n^3$
TMUL	$\frac{1}{2}n^2(n+1)$	$2n^2(n+1)$
DOT	n^2	$4n^2$
ADD	$2n^2$	$4n^2$
SCALE	n^2	$4n^2$
VMUL	n^2	$4n^2$
ABS2	n^2	$2n^2$

Table 3.2. Number of required DSP arithmetic instructions for matrix functions in $\mathbb{C}^{n\times n}$

The complexity of the CHOL and DIV instructions are also analysed. DSP instructions are categorised according to their type, depending on if they concern real or complex multiplication (*mpy*), multiply-accumulate (*mac*), multiply-subtract (*mas*), division (*div*), square-root (*sqr*), or inverse square-root (*isqr*). While the first instructions have a high throughput, the operations involving divisions and square-roots are likely to increase the latency of the functions.

When floating-point format is used, the algorithms can be slightly modified to limit the number of division operations which are time-consuming. For Cholesky decomposition in fixed-point, the real value l_{jj} is computed at first with the *sqr* operation, and then it is used to divide all the other coefficients in the same column. In floating-point, both the square-root and the inverse square-root of $a_{jj} - \sum_{k=0}^{j-1} l_{jk}\overline{l_{jk}}$ can be computed. The former is stored as the value of l_{jj} and the latter is multiplied to all the l_{ij} for $i > j$. Therefore no *div* instruction is needed in this implementation (see all results in Table 3.3).

Similarly, the Gaussian elimination and Gauss-Jordan algorithm can be modified in floating-point. Before eliminating the coefficients in the column of the pivot a_{kk}, the row k can be normalised by dividing it by a_{kk}. Instead of computing a division for all coefficients in the row, $1/a_{kk}$ is first computed and then multiplied to other data. This will avoid the computation of a_{ik}/a_{kk} for every i because a_{kk} has been normalised to 1. This algorithm is described in more details in Appendix B. The normalisation will also make the back-substitution faster (if Gaussian elimination is used), because all the diagonal coefficients will be equal to 1.

The theoretical results for DIV are given in Table 3.4. Some additional operations which are not taken into account here are required for the partial pivoting (compare the pivot and switch row). In the first version with mixed-use of fixed-point and floating-point computations, the data must also be converted before doing the back-substitution.

For the special case where $B \in \mathbb{C}^{n\times 1}$ (column vector) in the computation of

	Fixed-point	Floating-point
mas	$\frac{1}{6}(n^3 - n)$	$\frac{1}{6}(4n^3 - 6n^2 + 2n)$
sqr	n	n
isqr	0	n
div	$n^2 - n$	0
mpy	0	$n^2 - n$

Table 3.3. Number of required DSP instructions for CHOL function on matrix in $\mathbb{C}^{n \times n}$

	Gaussian elimination in fixed-point and back substitution in floating-point	Gaussian elimination and back substitution in floating-point	Gauss-Jordan elimination in floating-point
mpy (search pivot)	$\frac{1}{2}(n^2 + n)$	$n^2 + n$	$n^2 + n$
div (by pivot)	$2n^2 - 2n$	$2n$	$2n$
mpy (compute $\frac{a_{ik}}{a_{kk}}$ or normalise row)	$n^2 - n$	$6n^2 - 2n$	$6n^2 - 2n$
mas (eliminate coefficients)	$\frac{1}{6}(5n^3 - 6n^2 + n)$	$\frac{1}{3}(10n^3 - 12n^2 + 2n)$	$6n^3 - 8n^2 + 2n$
mas (back substitution)	$2n^3 - 2n^2$	$2n^3 - 2n^2$	0
div (back substitution)	$2n$	0	0
mpy (back substitution)	$4n^2$	0	0
Asymptotic complexity	$\Theta\left(\frac{17}{6}n^3\right)$	$\Theta\left(\frac{32}{6}n^3\right)$	$\Theta\left(\frac{36}{6}n^3\right)$

Table 3.4. Number of required DSP instructions for DIV function on matrix in $\mathbb{C}^{n \times n}$

$A^{-1}B$, the respective asymptotic complexities are $\Theta\left(\frac{1}{3}n^3\right)$ for mixed fixed-point / floating-point algorithm, $\Theta\left(\frac{4}{3}n^3\right)$ for floating-point Gaussian elimination, and $\Theta\left(\frac{6}{3}n^3\right)$ for floating-point Gauss-Jordan elimination.

It can be deduced from these results that the first version will probably be faster, especially when n is large, but it will be limited by all the *div* instructions. In pure floating-point implementation, both mentioned versions are worth considering in the general case, because the performance may be similar depending on how the instructions can be run in parallel each cycle in an optimised schedule. It can also be noted that if the input A is triangular and only the substitution step has to be computed, the execution will be much faster than in the general case (complexity in $\Theta(2n^3)$), and even more if B is a column vector (complexity in $\Theta(2n^2)$).

All the results in this section are theoretical results which aim at giving a rough idea of the latency of the matrix functions. Several factors such as instruction latency, limitations in the read/write operations, restricted number of registers, and branching may affect the speed of each function.

Part II

Implementation and Performance

4 Implementation

The actual implementation of the matrix algorithms will take into account all the principles described previously. The aim is to control the data format, data range, and matrices properties, in order to achieve the best possible performance. Trade-off must be made to find the right level of control: more control can lead to higher performance in terms of speed and accuracy, but the control operations themselves may generate latency (data conversion, evaluation of data range, etc.).

During implementation, the scheduling of the elementary instruction in the DSP programs will also be critical to achieve the best possible speed of execution.

In all this part, the finite representation of a number a will be written $\hat{a} = \widetilde{a} \times 2^{exp_a}$. $\widetilde{a}$ is the number stored in the $Qm.n$ fixed-point representation, with $\|\widetilde{a}\|_\infty \leq 2^{m-1}$. exp_a is an integer used to scale the number, and it will typically be common for all the coefficient of a matrix.

4.1 Information on stored matrices

In order to apply the right version of the matrix functions in each cases, it is necessary to store information about the matrices that are currently stored in *smx_mem*.

This information includes:

- The format: $Q1.15$ fixed-point or single-precision floating-point
- The shape: upper triangular, lower triangular, diagonal, ordinary
- Exponent (only valid in fixed-point)

The exponent exp_A of a matrix A can be interpreted as the following: for every i, j in $\{0, \ldots, n-1\}$, $\hat{a_{ij}} = \widetilde{a_{ij}} 2^{exp_A}$.

All this information about the matrices is stored in a data structure called *smx_mem_manager*. It is read before executing a matrix function and updated after the execution if information has changed. It is meant to be flexible, and to help to obtain the best possible performance for the matrix jobs required for the client. The updated global structure of the code is shown in Figure 4.1

A client which sends a job to a matrix server can require that all computations are done in floating-point. In this case, all matrices will be converted in the single-precision floating-point format as soon as they are loaded in *smx_mem*, and will

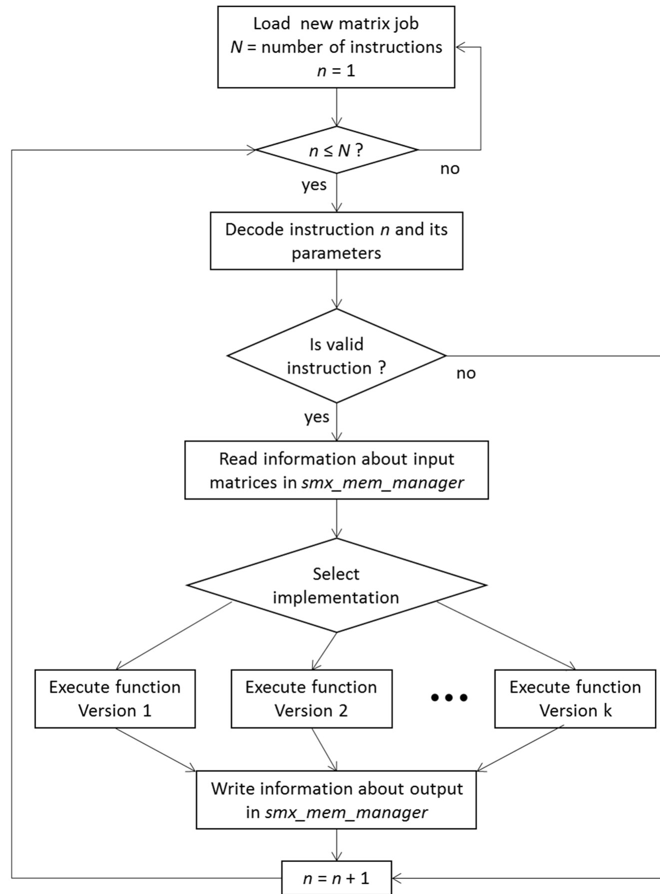

Figure 4.1. Updated structure of the code for a Matrix server DSP

remain in this format during the execution of the whole job. Otherwise the format may change between matrix instructions.

4.2 Control of data range in fixed-point

4.2.1 Bit shifts in multiplication

When two real numbers in $Q1.15$ format are multiplied ($c = ab$), the result of the product $\widetilde{a} \times \widetilde{b}$ is first obtained in $Q2.30$ format. Then two methods can be used to compute the final result $\hat{c} = \widetilde{c} \times 2^{exp_c}$:

- First method: The result is left-shifted by one bit, and in this case the result $\widetilde{c} = \widetilde{ab}$ is written in $Q1.31$ format, and $exp_c = exp_a + exp_b$. The 16 least significant bits can be dropped after rounding, if $\widetilde{c}$ has to be written in $Q1.15$ format.

- Second method: The $Q2.30$ result is not modified, but is interpreted as a $Q1.31$ number. In this case $\widetilde{c} = \widetilde{ab}/2$, and $exp_c = exp_a + exp_b + 1$. Again only the 16 most significant bits will be kept if $\widetilde{c}$ must be obtained in $Q1.15$ format.

Even though the first method would be more intuitive when real numbers are considered, the second one can be very useful for complex data. When computing the real and imaginary parts of $c = ab = (a_{re}b_{re} - a_{im}b_{im}) + i(a_{re}b_{im} + a_{im}b_{re})$, only the first method may generate overflow.

When n products are accumulated, it has already been proven that the final $Q1.15$ result will need to be right-shifted by $1 + \lceil \log_2(n) \rceil$ bits to avoid overflow. In practice, the partial accumulated results can be stored in 40-bit registers in $Q10.30$ format. At the end of the computation, the result will be right-shifted by only $\lceil \log_2(n) \rceil$ bits.

4.2.2 Exponent for the ADD instruction

The main difficulty of the addition in fixed-point is that the inputs must have the same exponent. For the ADD instruction, the matrix $A = A + \lambda B$ is computed. Algorithm 1 describes both how to align the exponents of A and λB and how to avoid overflows in the result. All inputs are assumed to be stored in $Q1.15$ format.

Note: in the algorithm, "$\widetilde{x} >> k$" means that $\widetilde{x}$ is right-shifted by k bits. This is done for both the real and imaginary parts if x is complex.

Algorithm 1 ADD $A = A + \lambda B$ with $\lambda \in \mathbb{C}$

if $exp_\lambda + exp_B + 1 \leq exp_A$ **then**
  $\widetilde{\lambda} \leftarrow \widetilde{\lambda} \gg exp_A - exp_\lambda - exp_B - 1$
  $exp_\lambda \leftarrow exp_\lambda + (exp_A - exp_\lambda - exp_B - 1) = exp_A - exp_B - 1$
  for $i, j = 0 \ldots n-1$ **do** ▷ $T = \lambda B$
    $\widetilde{t_{ij}} \leftarrow \widetilde{\lambda}\widetilde{b_{ij}}/2$ ▷ $exp_T = exp_\lambda + exp_B + 1 = exp_A$, $\left\|\widetilde{t_{ij}}\right\|_\infty \leq 1$
    $\widetilde{a_{ij}} \leftarrow (\widetilde{a_{ij}} + \widetilde{t_{ij}})/2$ ▷ $\|\widetilde{a_{ij}}\|_\infty \leq 1$
  $exp_A \leftarrow exp_A + 1$
else ▷ $exp_\lambda + exp_B + 1 > exp_A$
  if $\left\|\widetilde{\lambda}\right\|_\infty > 0.5$ **then**
    $\widetilde{\lambda} \leftarrow \widetilde{\lambda} \gg 1$
    $exp_\lambda \leftarrow exp_\lambda + 1$
  for $i, j = 0 \ldots n-1$ **do**
    $\widetilde{a_{ij}} \leftarrow \widetilde{a_{ij}} \gg exp_\lambda + exp_B + 1 - exp_A$ ▷ $\|\widetilde{a_{ij}}\|_\infty \leq 0.5$
    $\widetilde{a_{ij}} \leftarrow \widetilde{a_{ij}} + \widetilde{\lambda}\widetilde{b_{ij}}/2$ ▷ $\left\|\widetilde{\lambda}\widetilde{b_{ij}}/2\right\|_\infty \leq 0.5$, $\|\widetilde{a_{ij}}\|_\infty \leq 1$
  $exp_A \leftarrow exp_\lambda + exp_B + 1$

If λ is not well-conditioned at the beginning, in other words if $\left\|\widetilde{\lambda}\right\|_\infty$ is very small compared to 1, then too many significant bits may be lost in the second case of the algorithm ($exp_\lambda + exp_B + 1 > exp_A$). A solution is to left-shift $\widetilde{\lambda}$ as much as it is possible without generating overflow, or at least until $\left\|\widetilde{\lambda}\right\|_\infty \geq 0.25$.

This algorithm can be slightly modified when λ is a real number, or in the special cases where $\lambda = 1$ or $\lambda = -1$, and $exp_A = exp_B$.

4.2.3 Control of data range

If it has not been required by the client that all computations must be done in floating-point, fixed-point will be used in some functions. It is necessary to control the data range of the stored matrices both to avoid overflow and maintain a sufficient number of significant bits. The following rules can be used:

- When fixed-point data is loaded in *smx_mem*, it is directly converted in $Q1.15$ format. When floating-point data is loaded, it is first stored in single-precision floating-point format.

- Before the execution of all instructions except VINV and VSQRI, input matrices which were stored in floating-point are converted in fixed-point. For every matrix A in floating-point, an exponent exp_A is found such that $2^{exp_A - 1} \leq \max(\{\|\hat{a_{ij}}\|_\infty : i, j \in \{0, \ldots, n-1\}\}) < 2^{exp_A}$, and then $2^{-exp_A} \times A$ is converted in $Q1.15$ format.

- For all the instructions which involve at the most one complex multiplication per coefficient (SCALE, VMUL, ABS2, and DMAX), the numbers are multiplied without any shifts in the result. This gives for instance the following output C for the elementwise multiplication of A and B: $\forall i, j \in \{0, \ldots, n-1\}$, $\widetilde{c_{ij}} = \widetilde{a_{ij}}\widetilde{b_{ij}}/2$, and $exp_C = exp_A + exp_B + 1$.

- For the instructions with product accumulation (MUL, TMUL, and DOT), the results are right-shifted as described previously. The matrix multiplication $C = AB$ gives for instance:

$$\forall i, j \in \{0, \ldots, n-1\},\ \widetilde{c_{ij}} = \sum_{k=0}^{n-1} \widetilde{a_{ik}}\widetilde{b_{kj}} \times 2^{-1-\lceil \log_2(n) \rceil}$$

  and $exp_C = exp_A + exp_B + 1 + \lceil \log_2(n) \rceil$

- The ADD instruction is executed following Algorithm 1.

- Before executing Gaussian elimination algorithm in fixed-point, both input matrices are scaled by $2^{-1-\lceil \log_2(n) \rceil}$ in order to avoid overflow during the computations.

With this method, there is a risk that all significant bits are lost after several consecutive matrix instructions. Always considering the worst case for the risk of overflow may generate too many right shifts than actually necessary in the result. One solution would be to keep track of the maximum norm of matrix coefficients.

The number $lmax$ defined as followed for every matrix, can be used to evaluate data range of $Q1.15$ matrix coefficients:

$$lmax_A = -\max(\{\lceil \log_2 (\|\widetilde{a_{ij}}\|_\infty) \rceil : i, j \in \{0, \ldots, n-1\}\})$$

In other words, $lmax$ is the greatest integer such that $\forall i, j \in \{0, \ldots, n-1\}, \|\widetilde{a_{ij}}\|_\infty \leq 2^{-lmax}$ (undefined when $A = 0$, this case is ignored here).

If this number is computed for input A and B when computing $C = AB$, the final right shift of the result will be of $\lceil \log_2 (n) \rceil - lmax_A - lmax_B$ bits instead of $\lceil \log_2 (n) \rceil$. It is assumed that a right shift of a negative k number of bits is the same a left shift of $-k$ bits.

This evaluation will be done at the beginning of MUL, TMUL, and DOT instruction, because they are the ones that may generate the most important loss of significant bits. Algorithm 2 describes the whole execution of the MUL instruction.

The number $lmax$ is also computed for both inputs A and B before the fixed-point Gaussian elimination step in the DIV instruction ($C = A^{-1}B$). In order to limit the loss of significant bits, they are respectively scaled by $2^{-1-\lceil \log_2(n) \rceil + lmax_A}$ and $2^{-1-\lceil \log_2(n) \rceil + lmax_B}$ before any other operation.

Algorithm 2 MUL $C = AB$

function MUL_PREPARE_INPUT(A)
  $lmax_A \leftarrow -\max(\{\lceil \log_2 (\|\widetilde{a_{ij}}\|_\infty) \rceil : i, j \in \{0, \ldots, n-1\}\})$
  if A is stored in floating-point **then**
    Convert $2^{lmax_A} \times A$ in $Q1.15$ format
    $exp_A \leftarrow -lmax_A$
    $lmax_A \leftarrow 0$ ▷ Data converted with exponent such that final $lmax_A = 0$
end function

function MUL(A, B)
  MUL_PREPARE_INPUT(A)
  MUL_PREPARE_INPUT(B)
  $rs = 1 + \lceil log_2(n) \rceil - lmax_A - lmax_B$
  for $i, j = 0 \ldots n-1$ **do**
    $\widetilde{c_{ij}} \leftarrow 0$
    for $k = 0 \ldots n-1$ **do**
      $\widetilde{c_{ij}} \leftarrow \widetilde{c_{ij}} + \widetilde{a_{ik}} \widetilde{b_{kj}}$ ▷ $\|\widetilde{c_{ij}}\|_\infty \leq 2^{rs}$
    $\widetilde{c_{ij}} \leftarrow \widetilde{c_{ij}} >> rs$
  $exp_C = exp_A + exp_B + rs$
end function

4.3 Load and store instructions

As explained in the previous chapter, the LOAD instruction cannot be implemented simply by sending a few requests to the shared memory to load data directly into *smx_mem*. Sending typically one request per row would be too costly, and would not be relevant for instance if the destination matrix must be transposed. Instead an intermediate buffer must be used to load/store larger sets of data.

The DSP load instructions are non-blocking, and data received at the beginning of a read request can be used in computations before the last data has been received. A possible way to use this feature is to use two buffers, which can each contain around a few hundreds of matrix coefficients. The data will be loaded alternatively in the two buffers, and then copied, converted, transposed, and/or conjugated into *smx_mem*.

Each buffer can contain a group of a given number of rows, whose size depends on the buffer size and the matrix dimensions. Algorithm 3 describes how to make use of the non-blocking load requests.

Algorithm 3 LOAD / LOAD_T / LOAD_C / LOAD_TC

Load first group of rows in buffer 0
$k \leftarrow 1$
while Number of remaining rows to load > 0 **do**
 Load $(k+1)^{th}$ group of rows in buffer $((k+1) \bmod 2)$
 Copy k^{th} group of rows from buffer $(k \bmod 2)$ to *smx_mem*, convert / conjugate / transpose in place in necessary
 $k \leftarrow k+1$
end while
Copy k^{th} group of rows (last group) from buffer $((k-1) \bmod 2)$ to *smx_mem*, convert / conjugate / transpose in place in necessary

Figure 4.2 shows the schedule for the LOAD of a 10×10 matrix. Each buffer can contain up to 3 rows. The latency due to the non-blocking part of the load operation is partly hidden thanks to the use of 2 different buffers. In this case the copy rate is supposed to be higher than the rate at which data is loaded. Coefficients are therefore copied little by little as soon as they have been received in the buffer (step "Load wait + Copy").

A same similar algorithm can be used for the STORE instruction, the data is in this case first copied from *smx_mem* to the buffer, and then stored in the shared memory. VLOAD and VSTORE instructions can also be implemented in a similar way in the general case.

When data is not transposed or conjugated, and when both external and internal format is complex single-precision floating-point, there is a much simpler implementation for VLOAD and VSTORE: data is directly loaded/stored to/from *smx_mem*, there is no need to use an intermediate buffer. Because the load/store requests sent by the DSP are non-blocking, the DSP can simply send a succession

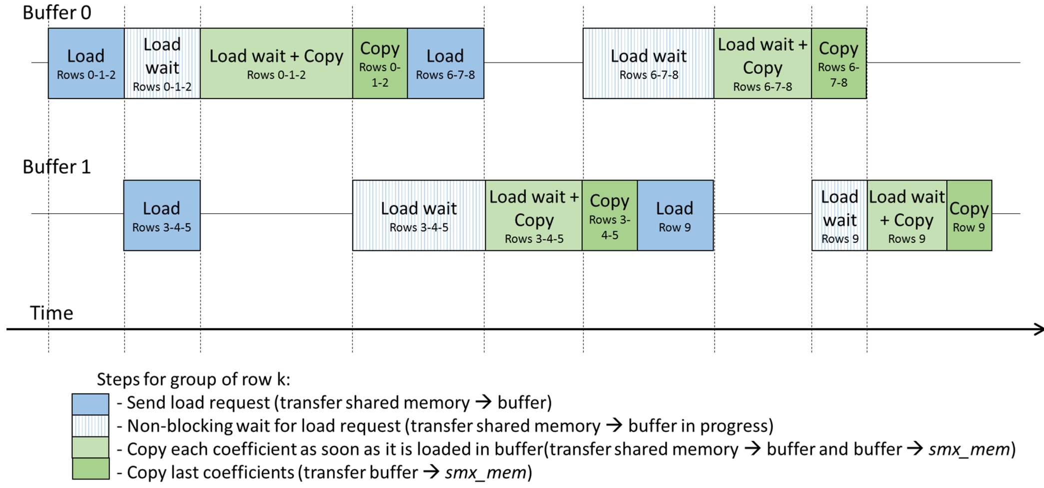

Figure 4.2. Load schedule for 10×10 matrix

of requests, and then begin the execution of the next matrix instruction while data is still transferred.

4.4 Schedule and loop unrolling

In order to obtain an optimal schedule for each function, up to 4 DSP instructions must be executed in parallel at each cycle. These instructions must have different outputs, which implies that several coefficients of the resulting matrix must be computed at the same time.

An important assumption in all functions except for the load and store is that all the inputs and outputs are square matrices of the same size, and this size is a multiple of 4. This is true thanks to the specification of the LOAD instruction: during the load, rectangular matrices of any size can be considered in the shared memory, but in the local memory they will be placed in a square block of size $4k \times 4k$, and the rest of the block that does not contain data is filled with 0s. It is this full block that will be considered as a matrix in the rest of the matrix job.

A possible way to unroll the loops in the matrix functions is to compute 4 consecutive coefficients in the same row of the output matrix. Other loops can also be unrolled to limit the latency overhead due to set up of the loops (update address pointers, set number of iterations, read first data, write last data, etc.) A possible schedule for MUL is described in Algorithm 4. When the index $(4j \ldots 4j+3)$ is used in one instruction, it means that four instructions are executed in parallel for the indexes $4j$, $4j+1$, $4j+2$, and $4j+3$.

Another way to unroll the loops can be to consider 2×2 blocks, instead of 4 consecutive coefficients. This can be useful especially when the diagonal is im-

Algorithm 4 Simplified schedule of MUL: $C = AB$ in fixed-point

for $i = 0 \ldots n-1$ **do**
  for $j = 0 \ldots \frac{n}{4} - 1$ **do**
    $c_{(i)(4j \ldots 4j+3)} \leftarrow 0$
    for $k = 0 \ldots \frac{n}{4} - 1$ **do**
      $c_{(i)(4j \ldots 4j+3)} \leftarrow c_{(i)(4j \ldots 4j+3)} + a_{(i)(4k)} b_{(4k)(4j \ldots 4j+3)}$
      $c_{(i)(4j \ldots 4j+3)} \leftarrow c_{(i)(4j \ldots 4j+3)} + a_{(i)(4k+1)} b_{(4k+1)(4j \ldots 4j+3)}$
      $c_{(i)(4j \ldots 4j+3)} \leftarrow c_{(i)(4j \ldots 4j+3)} + a_{(i)(4k+2)} b_{(4k+2)(4j \ldots 4j+3)}$
      $c_{(i)(4j \ldots 4j+3)} \leftarrow c_{(i)(4j \ldots 4j+3)} + a_{(i)(4k+3)} b_{(4k+3)(4j \ldots 4j+3)}$

portant in the computation. For instance in the TMUL instructions, $C = AA^*$ is computed. The result is a Hermitian matrix, which means that only the coefficients on and above the main diagonal need to be computed, and then the coefficients are conjugated and written in the other half of the matrix (see a simplified schedule in Algorithm 5 and Figure 4.3).

Algorithm 5 Simplified schedule of TMUL: $C = AA^*$

for $i = 0 \ldots (\frac{n}{2} - 1)$ **do** ▷ Diagonal block first

  Compute $c_{(2i \ldots 2i+1)(2i \ldots 2i+1)} = \sum_{k=0}^{n-1} a_{(2i \ldots 2i+1)(k)} \times conj(a_{(2i \ldots 2i+1)(k)})$

  for $j = (i+1) \ldots (\frac{n}{2} - 1)$ **do** ▷ Other blocks

    Compute $c_{(2i \ldots 2i+1)(2j \ldots 2j+1)} = \sum_{k=0}^{n-1} a_{(2i \ldots 2i+1)(k)} \times conj(a_{(2j \ldots 2j+1)(k)})$

    Compute $c_{(2j \ldots 2j+1)(2i \ldots 2i+1)} = conj(c_{(2i \ldots 2i+1)(2j \ldots 2j+1)})$

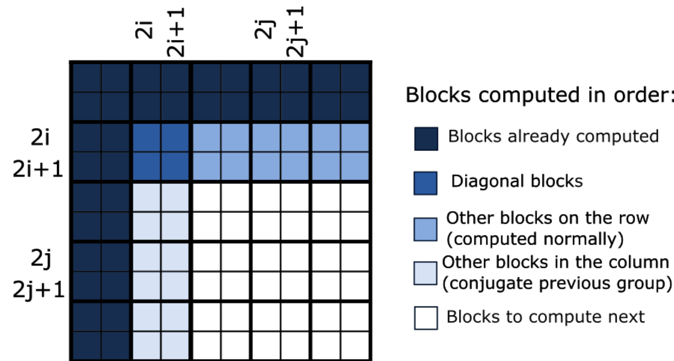

Figure 4.3. Output matrix of TMUL - Decomposition in blocks

5 Evaluation

For the precision and speed analysis, the algorithms are executed on a cycle-accurate simulator, which emulates the behaviour of the hardware platform while some features are still under development. The cycle-accurate simulator enables a precise evaluation of the speed of each function. Only a simplified version of the platform containing one server DSP and the shared memory is emulated. For the hardware accelerator, the latency values were already available at Ericsson based on previous simulations.

The output of the matrix program can also be obtained with the simulator, by displaying the content of the shared memory at the end of the simulation. In order to evaluate the accuracy of the results, these values will be compared with the results obtained with the Matlab software. The latter uses double-precision floating-point format for the computations, so these results may also contain errors but they are hopefully closer to the exact results.

For the power analysis, the simulations have been run on Register Transfer Level (RTL) and gate level simulators, which record the activity in the architecture during the whole execution. The results of these simulations have then been analysed with an Ericsson internal tool. This tool measures the switching activity, that is to say the rate at which the values change in the circuits (from 0 to 1 or from 1 to 0). This activity is measured over time for each sub-components of the architecture. It is for instance possible to visualise the activity of different units inside the DSP, from the instruction fetch unit to the floating-point computational units, during the execution of each function.

5.1 Evaluation of instructions

First, the accuracy and speed of the computations have been tested for matrix jobs which contain only one operation. The matrix jobs that are built contain one or several LOAD instructions for the inputs, then one actual operation, and then one STORE instruction to write the result in the shared memory.

5.1.1 Accuracy

Generation of input matrices

In order to evaluate the accuracy of the computations, random matrices have been generated with Matlab, in single precision floating-point format, and in $Q1.15$ fixed-point format. All the matrices contain complex data, and the real parts and imaginary parts of each coefficient have a uniform distribution inside the range [-1,1].

It can be noted that with this distribution, data range might be wide, but remains homogeneous: approximately 50% of real an imaginary parts absolute values are in the range $[0.5, 1]$, 75% in the range $[0.25, 1]$, etc. This ensures that a number of significant bits are provided for most coefficients.

For the CHOL instruction, the input matrices need to be Hermitian definite positive. First, a lower triangular matrix L is generated. The diagonal coefficients are real values in the range $[0.5, 1]$, and the complex coefficients below the diagonal have their real and imaginary parts in the interval $[-0.5, 0.5]$. All the distributions are uniform in their respective range. Then the matrix $A = LL^*$ is computed, scaled so that all its coefficients fit in the interval $[-1, 1]$, and converted after rounding in one of the input formats. The intervals in L have been chosen so that the diagonal coefficients are much larger than the others regarding infinite norm. Otherwise the matrix A is not always considered as positive definite once it is converted in $Q1.15$ format, and the Cholesky decomposition fails.

The finite representation of the data in the given input formats are assumed to be perfect, i.e. there is no error on the input matrix loaded from the shared memory (this may not be true in practice if these data are obtained after measurements rounding or are the results of previous computations in finite formats).

In these tests, all the inputs and outputs are in the same format, called *external format*. Internal computations are either done in fixed-point or in floating-point after a potential conversion, and then data are converted again in the external format before or during the execution of the STORE function.

Scaling of output fixed-point matrices

When the output format is $Q1.15$ fixed-point, the final result of the $n \times n$ matrix is scaled to avoid overflow, for instance by a factor $2^{-(1+\lceil \log_2(n) \rceil)}$ for MUL and by a factor 2^{-1} for ADD.

For the division, it is not possible to guarantee the absence of overflow in fixed-point with a given scaling factor, because there is no bound on the coefficients of the matrix $C = A^{-1}B$ when random matrices A and B are considered (and this was the initial reason why the last steps of the DIV operation is always internally executed in floating-point).

However, when the real and imaginary parts of the inputs' coefficients have a uniform distribution inside the range $[-1, 1]$, it is possible to find a reasonable bound which is valid not for all but for most output coefficients. Two sets (A_i) and (B_i)

of 10,000 such random matrices have been generated in Matlab, and the range of the matrices $C_i = A_i^{-1}B_i$ have been evaluated.

It has been obtained experimentally that for more than 99.5% of the coefficients of the output matrices, both the real and imaginary parts are in the range $[-2^4, 2^4]$, and this is true for all the considered matrix sizes. The scaling factor 2^{-4} has therefore been used before writing the result of the DIV operations in the $Q1.15$ fixed-point output format. The potential out-of-bounds coefficients will saturate and may generate an important error.

Evaluation of output

The outputs of the matrix software are compared to the output obtained with Matlab for the same operation. The relative error between the reference Matlab matrix M and the tested matrix T is defined by the average of the relative error for every coefficient:

$$\rho = \frac{1}{n^2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \frac{|t_{ij} - m_{ij}|}{|t_{ij}|}$$

When some of the coefficients of the output matrix are forced to 0 by the definition of the function, these values will not be taken into account in the error. For instance, the output of CHOL is a lower triangular matrix, and only the coefficients on and below the diagonal are considered in the average value:

$$\rho_{\text{CHOL}} = \frac{2}{n(n+1)} \sum_{i=0}^{n-1} \sum_{j=0}^{i} \frac{|t_{ij} - m_{ij}|}{|t_{ij}|}$$

For the DIV function with $Q1.15$ fixed-point output format, only the coefficients that have not generated overflows are taken into account in the average error value. If such output coefficients outside of the range $[-1, 1]$ after scaling were included in the average computation, large error may be obtained for the matrices which contain such coefficients. The results that are presented below aim at presenting the general performance of the software program, in a more realistic use it would be the responsibility of user to use the DIV function with fixed-point output format only for well-conditioned matrices if these overflows are not tolerated.

For each configuration given by the function type, matrix size, internal and external format, 10 tests have been run with different inputs. The results that are given below are computed as the average value of the errors obtained with the 10 output matrices.

Results

The results that are obtained for the MUL instruction are shown in Figure 5.1.

Internal format \ External format	Floating-point	Fixed-point
Floating-point	$2^{-25.5}$	$2^{-14.7}$
Fixed-point	$2^{-13.2}$	$2^{-13.3}$

Table 5.1. Error on output for ADD instruction

It can be noticed that there is a significant gain in precision when floating-point is both the external and the internal format. When the external format is $Q1.15$ fixed-point, then the accuracy is only slightly better when the computations are done in floating-point, the precision is similar: in floating-point, a few significant bits are lost during the multiply-accumulate operations, and the error is dominated by the rounding due to the final scaling and conversion into $Q1.15$ format. In fixed-point, no significant bit is lost during the multiply-accumulate operations in 40-bit registers, but error is also introduced with the final scaling and the final write in 16+16-bit format.

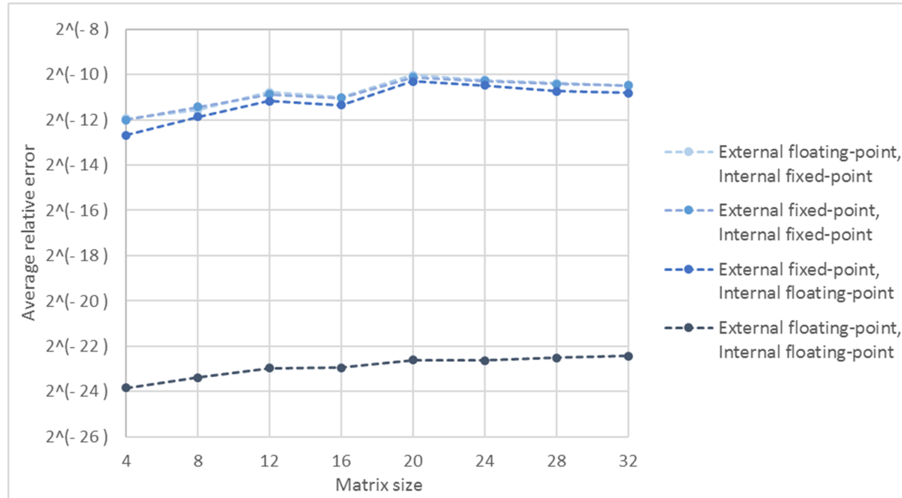

Figure 5.1. Error on output of MUL instruction

For the ADD instructions, the matrix $C = A + \lambda B$ has been computed with $\lambda = 1$. The average error does not depend on the size of the matrix and is shown in Table 5.1. Again, the precision is much higher if both external and internal formats are floating-point. If external format is fixed-point, the accuracy of the result is only slightly increased when using internal floating-point computations.

For the DIV instruction, the algorithm that is used here is the Gaussian elimination followed by a back-substitution. The internal format that is mentioned is the format for the Gaussian elimination, since the second step is always executed in floating-point.

It can be noticed with the results shown in Figure 5.2 that the best precision is obtained again when both external and internal data are always represented in floating-point. It can be noticed however that even with these formats the error grows more with the size of the matrix than for MUL.

When the external format is $Q1.15$ and the internal format is floating-point the error does not vary significantly with the size of the matrix. This suggests that the error is mostly due to the final scaling of the result by the constant factor 2^{-4}. The effects of the internal computations seem to be negligible in this case.

Using fixed-point for the Gaussian elimination generates important error, especially for the largest matrices. The initial scaling of the inputs that is done to avoid overflow during the elimination seems to be one of the main causes.

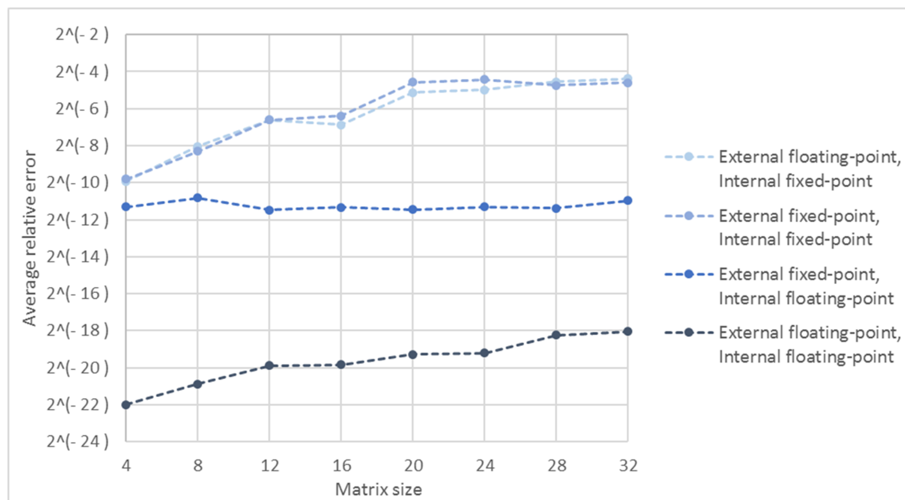

Figure 5.2. Error on output of DIV instruction

Similar remarks can be made about the results for the Cholesky decomposition (see Figure 5.3). The accuracy is satisfying for smaller matrices but grows quickly with the matrix size. When the external $Q1.15$ format is used, the gain when using the internal floating-point implementation is small for matrices containing up to 8×8 coefficients but is more important beyond that limit.

Conclusion about precision

To conclude, the precision depends greatly on the choice of internal and external formats. For DIV and CHOL, the internal fixed-point implementations can be used

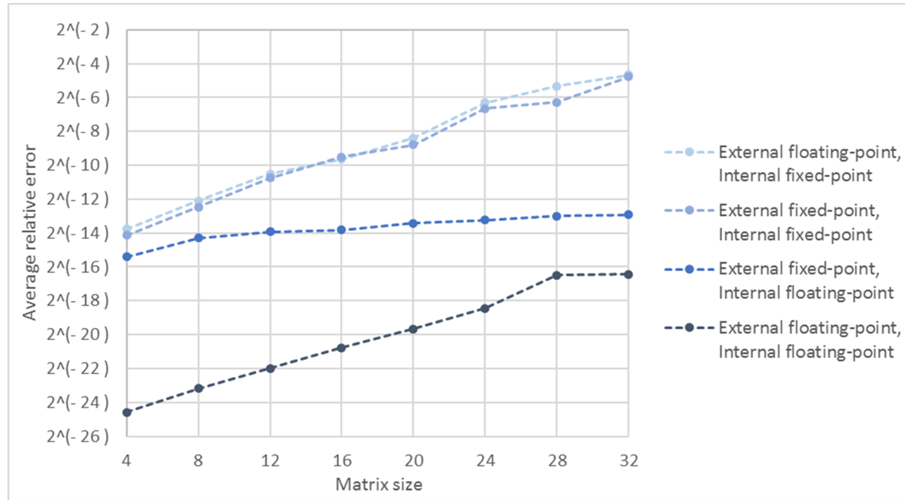

Figure 5.3. Error on output of CHOL instruction

when either the precision is not the priority of the user, or for small matrices when the external data is also stored in fixed-point.

Given the low precision obtained with the largest values of n in fixed-point for some functions, maybe three different internal modes should be implemented in the program:

- First mode: force all the internal computations in floating-point
- Second mode: force all the internal computations in fixed-point except when it is not possible (exception for VINV, VSQRI, end of DIV)
- Third mode: force internal computations in fixed-point for the most stable functions only (not for DIV and CHOL)

For the moment only the two first modes have been implemented. The choice of the mode is the responsibility of the user. The results in this report can be given as indications, and further tests for the specific problem can help the user to take the best decisions.

Now the latencies of the different implementations need to be compared to evaluate what can be the advantages of selecting the least accurate versions of the functions.

5.1.2 Instruction latency

The latency of execution has been measured for every instruction. The execution of an instruction can be divided into several steps:

1. Instruction decoding and parameters check (all implementations)
2. Preparation of inputs (only software implementations)
3. Actual computations (all implementations)
4. Matrix information update (only software implementations)

All these steps are taken into account in the latency measurements.

In the hardware accelerator, the preliminary first step can in most cases operate partly in parallel with the execution of the previous instruction, so it does not generate an important additional delay. The total instruction latency for the hardware therefore corresponds mostly to the actual matrix computations.

In the software however, the first step increases the total duration of the instruction. The operations of step 1 as well as operations of step 4 are written in C language, and are therefore not fully optimised when executed by the DSP. This overhead does not depend on the matrix size.

For the software implementation, the second step includes reading information about the inputs and choosing the right version of the functions according to the formats and shapes of the matrices. For some functions like MUL and DIV, this step in the fixed-point version also includes the evaluation of data range in the inputs to scale the values accurately in the computations (see description of the function in the previous chapter). This last function is written in assembly.

The experimental results obtained with the software are also compared with the ideal values, defined as the minimum latency required to execute only the computation step in a fully optimised way (four operations per cycle in the DSP instruction set). The read, write, and branching instructions are ignored in the ideal latency.

When comparing the results between the software and the hardware implementations, it must be kept in mind that the computational units are not the same in the DSP and in the accelerator. For floating-point computations, the throughput of the most common mathematical operations is about four times higher in the accelerator. When fixed-point is used in the software solution however, the main operations are only one to two times slower than in the accelerator, which always uses floating-point.

MUL

The results obtained for the MUL instruction are shown in Table 5.2 and Figure 5.4.

Even though the hardware accelerator is faster for all matrix sizes, the fixed-point version in software has similar performance for the biggest matrices. For smaller matrices, the cost of instruction fetching and input preparation adds a significant overhead, which makes this version the slowest. The floating-point implementation is approximately 4 times slower than the hardware, which confirms what has been obtained in the theoretical analysis.

Matrix size	Hardware accelerator	Software fixed-point: Experimental latency (Ideal latency)	Software floating-point: Experimental latency (Ideal latency)
4	56	342 (16)	238 (64)
8	168	560 (128)	734 (512)
12	700	1030 (432)	2022 (1728)
16	1190	1848 (1024)	4486 (4096)
20	2709	3110 (2000)	8510 (8000)
24	3822	4912 (3456)	14478 (13824)
28	6839	7350 (5488)	22774 (21952)
32	8855	10520 (8192)	33782 (32768)

Table 5.2. Latency of MUL instruction in cycles, experimental and ideal values

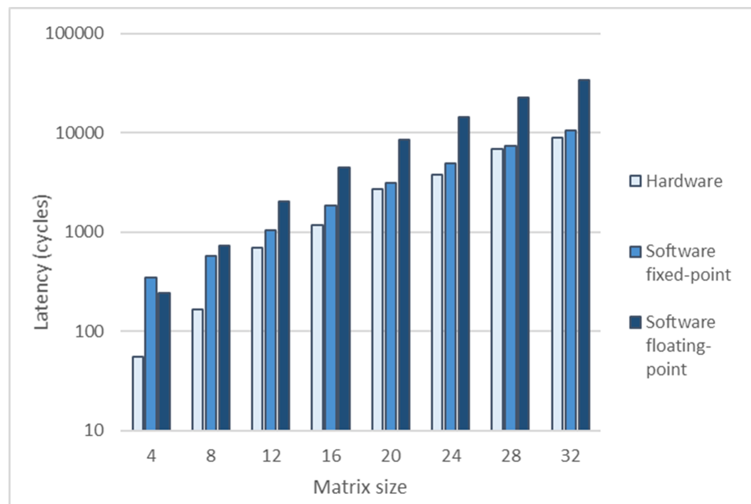

Figure 5.4. Comparison of experimental latency for MUL instruction

ADD

If only the computation step is considered, the fixed-point implementation has similar performance with the hardware accelerator for the ADD instruction. Because of the overhead due to other steps, this software version is from 1.5 to 6 times slower then the hardware. The floating-point version is only about 2.5 times slower for most sizes (see Table 5.3 and Figure 5.5).

Matrix size	Hardware accelerator	Software fixed-point: Experimental latency (Ideal latency)	Software floating-point: Experimental latency (Ideal latency)
4	28	175 (8)	149 (16)
8	77	203 (32)	201 (64)
12	112	247 (72)	285 (144)
16	168	307 (128)	383 (200)
20	231	383 (200)	549 (400)
24	315	475 (288)	729 (576)
28	399	583 (392)	941 (784)
32	504	707 (512)	1185 (1024)

Table 5.3. Latency of ADD instruction in cycles, experimental and ideal values

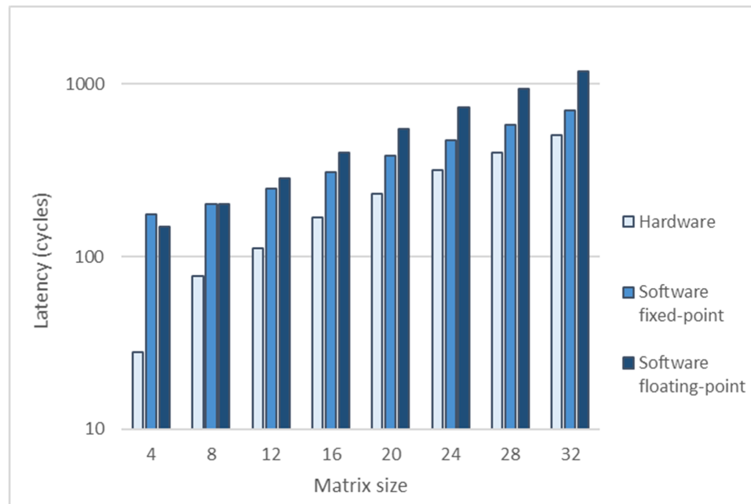

Figure 5.5. Comparison of experimental latency for ADD instruction

DIV

The latency of the DIV instructions has been evaluated for the three software implementations:

- Gaussian elimination (GE) in fixed-point + Conversion + Back substitution (BS) in floating-point
- Gaussian elimination (GE) + Back substitution (BS) both in floating-point
- Gauss-Jordan (GJ) algorithm in floating-point

Matrix size	Hardware accelerator	Software GE fixed-point + BS floating-point : Experimental latency (Ideal latency)	Software GE + BS floating-point: Experimental latency (Ideal latency)	Software GJ floating-point: Experimental latency (Ideal latency)
4	308	749 (83)	533 (97)	605 (101)
8	791	1833 (514)	1797 (714)	2247 (770)
12	1967	3915 (1565)	4486 (2363)	5730 (2583)
16	3276	7469 (3508)	9198 (5556)	11778 (6116)
20	6251	12859 (6615)	16576 (10805)	21162 (11945)
24	8953	20438 (11158)	27193 (18622)	34583 (20646)
28	14511	30638 (17409)	41576 (29519)	52696 (32795)
32	19187	43797 (25640)	60404 (44008)	76308 (48968)

Table 5.4. Latency of DIV instruction in cycles, experimental and ideal values

All these implementations have a variable latency due to the partial pivoting technique and the row swap which is executed only at some iterations of the main loop, depending on where the greatest pivot is found. The values presented in Table 5.4 and Figure 5.6 are the average latencies obtained with the 10 tests.

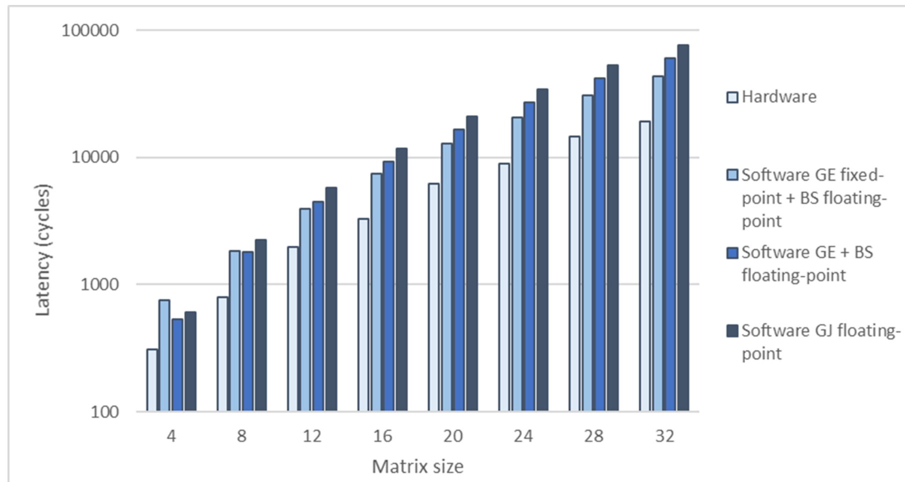

Figure 5.6. Comparison of experimental latency for DIV instruction

First, all the software implementations are at least 2 times slower than the hardware accelerator. The experimental values are also much higher than the ideal values even for big matrices, due to the fact that the algorithms cannot easily be optimised for the DSP (not always possible to unroll the loops and execute 4 useful

operations per cycle in practice).

If internal floating-point format is used, Gaussian elimination should be preferred to Gauss-Jordan algorithm, since the speed difference is even higher for the actual implementation than for the ideal one. Using fixed-point for the first step of the division increases the speed of the function, but the important loss of precision must be kept in mind when choosing this implementation.

The results that are given here are only for the general case, when the matrix for which the inverse is to be computed can have any shape. When this matrix is for instance an upper- or a lower-triangular matrix (result of a Cholesky decomposition for instance), the algorithms that are selected in software are respectively the backward-substitution and forward-substitution algorithms. These are much faster than the algorithms used for the general case, and they can only be executed in floating-point. In these specific cases, the software is up to 30% faster than the hardware.

CHOL

For the Cholesky decomposition, the software implementations are up to 50% slower than the hardware (see Table 5.5 and Figure 5.7). The two software versions have similar latencies, the floating-point being slightly faster for matrix sizes lower or equal to 20, and a bit slower for larger sizes. The large difference between the ideal and the experimental latency in fixed-point can partly be explained by the numerous division operations in this version: the long latency of these operations can in theory be hidden by doing other computations while waiting for the results, but in practice it cannot be done completely in a realistic schedule.

Together with the precisions results, these values suggest that the floating-point version should be preferred in general for the CHOL instruction.

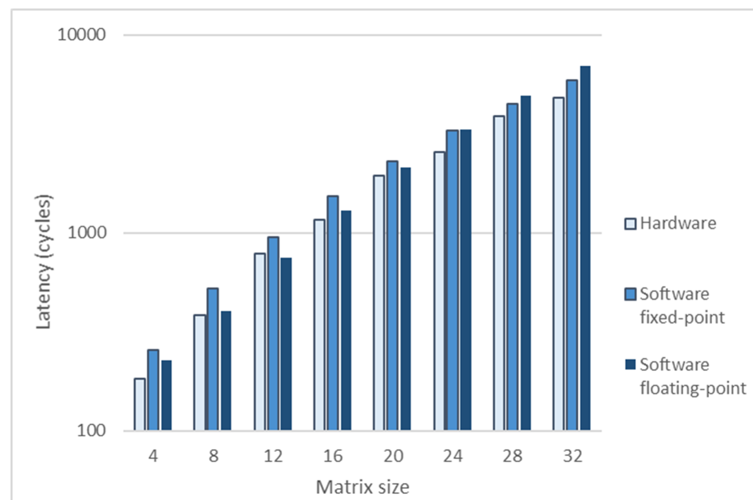

Figure 5.7. Comparison of experimental latency for CHOL instruction

Matrix size	Hardware accelerator	Software fixed-point: Experimental latency (Ideal latency)	Software floating-point: Experimental latency (Ideal latency)
4	182	257 (15)	227 (14)
8	385	525 (77)	402 (92)
12	784	948 (204)	747 (298)
16	1162	1524 (410)	1299 (696)
20	1946	2292 (713)	2129 (1350)
24	2562	3266 (1127)	3316 (2324)
28	3899	4454 (1670)	4915 (3682)
32	4823	5876 (2356)	6990 (5488)

Table 5.5. Latency of CHOL instruction in cycles, experimental and ideal values

5.2 Evaluation of a matrix algorithm

Several algorithms that involve matrix computations are used at Ericsson to improve the quality of the network signals, by limiting errors due to signal interference. The algorithm that is studied in more details below is a general pattern that appears with small modifications in several existing algorithms.

5.2.1 Jobs Description

The inputs of the algorithm are grouped in n vectors containing each k samples. The n vectors are stored consecutively in the shared memory. The whole set of data can be interpreted as a $n \times k$ rectangular matrix A.

The algorithm can be divided into two jobs. The aim of the first job is to build a covariance matrix R based on the input A, with the following steps.

1. VLOAD the n vectors of k data, filling $p = \lceil k/n \rceil$ matrices of size $n \times n$ in the internal memory.
2. TMUL for each of the p matrices
3. ADD all the p matrices together, to obtain the final $n \times n$ covariance matrix $R = AA^*$
4. SCALE R by $1/k$
5. STORE R in the shared memory

In the second job, the covariance matrix is used to update the initial data in A:

1. LOAD R from the shared memory
2. CHOL Compute the lower triangular matrix L such that $R = LL^*$
3. VLOAD the n vectors of k data (fill p matrices of size $n \times n$)
4. DIV Compute $L^{-1}A$ with DIV instructions for each of the p internal square matrices
5. VSTORE the modified n vectors of k data in the shared memory

Due to the limitations of size in the internal memory, all the input data cannot be loaded at the same time. The jobs are in this case composed of several iterations, and only part of the data is processed in each iteration.

When executed by the hardware accelerator, the second job can be optimised when $p > 1$, by using only one DIV instruction to compute L^{-1}, and then p MUL instructions to compute $L^{-1}A$.

In the software implementation however, the DIV instruction is faster than MUL: the fact that the output L of the CHOL instruction is a lower triangular matrix is saved in *smx_mem_manager*, and the algorithm that is chosen for all the following DIV instructions is therefore the forward-substitution algorithm, which is faster than the Gaussian elimination.

5.2.2 Tests

Floating-point input data have been generated for $n = 4, 8, 16,$ and 32, with 256 samples per vector. Each coefficient $a_{rj} = a_{rj_{norm}} \exp^{ia_{rj_{exp}}}$ is randomly generated. The norms $a_{rj_{norm}}$ are the absolute values of random numbers generated with a normal distribution bounded in the interval $[-1, 1]$. The complex exponents $a_{rj_{exp}}$ are uniformly distributed in $[0, 2\pi]$.

For each size n, a test has been run for every $k \geq n$ in the set {4, 8, 16, 32, 64, 128, 256}. Both internal formats have been tested, but the DIV instructions are always executed in floating-point (the forward-substitution algorithm contains division operations with no assumption on data range). All the output data is in floating-point format.

The accuracy of the output data has been evaluated with Matlab, and the latencies of the software implementations are compared with the latencies that would be obtained with the hardware accelerator.

5.2.3 Precision results

The average relative error on the $n \times k$ output of the algorithm is shown in Figure 5.8. The precision is again better with internal floating-point computations.

The error grows with n, but in general decreases with k, at least when k is close to n. This can be explained by the fact that when k is closed to n, the covariance

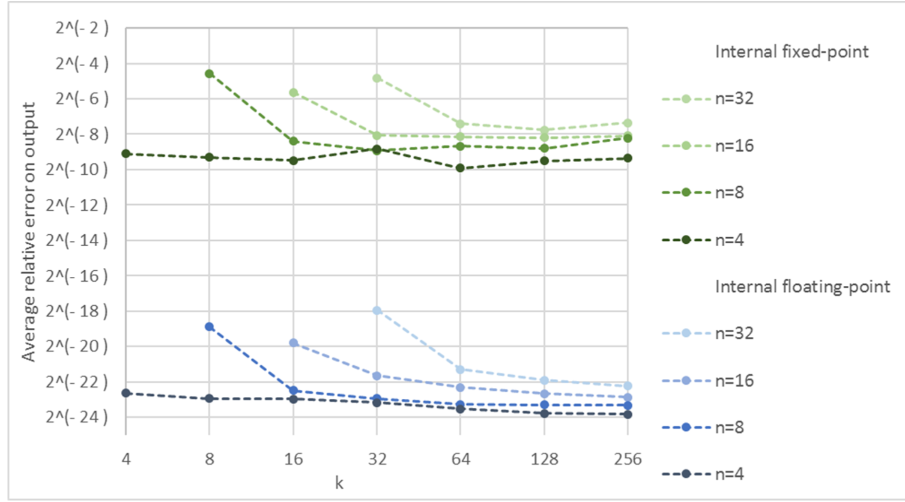

Figure 5.8. Error on output of matrix job

matrix is statistically ill-conditioned. The covariance matrix $R = AA^*$ is always Hermitian positive definite when A is invertible, but some of its eigenvalues are more likely to be close to zero when $k = n$ or $k = 2n$.

In fixed-point however, the error stops decreasing and sometimes slightly increases with larger k. This is due to the error on the covariance matrix, which grows with k in fixed-point. During the execution of the first job, several matrices are added together to build the covariance matrix. The heuristics that is used in fixed-point to avoid overflow during additions is in general pessimistic, and after a few ADD instructions with the same destination matrix, the number of significant bits is drastically reduced.

What happens more precisely is that for the destination matrix R with the coefficients $\hat{r_{ij}} = \widetilde{r_{ij}} \times 2^{exp_R}$, the finite representation of the coefficients $\widetilde{rij}$ gets very small, while the exponent exp_R gets very large. A solution could be to modify the implementation to evaluate the norm of the matrix coefficients $\widetilde{r_{ij}}$ every few instructions. If the maximum infinite norm is too small, the data could be left-shifted, and the exponent decreased accordingly. Even though this would globally increase the precision of the software, this solution would also increase the latency of the instructions.

5.2.4 Latency results

The cumulated latency of the two matrix jobs has been evaluated, and compared with the latency obtained with the hardware accelerator. The following value is computed for every combination of n and k:

$$\frac{\text{Latency of job 1 in software} + \text{Latency of job 2 in software}}{\text{Latency of job 1 in hardware} + \text{Latency of job 2 in hardware}}$$

n \ k	4	8	16	32	64	128	256
4	5.1	4.4	4.2	4.0	3.8	3.7	3.7
8		3.7	3.1	2.9	2.7	2.6	2.5
16			2.4	2.0	2.0	2.0	2.2
32				1.8	1.6	2.0	2.2

Table 5.6. Ratio: Latency software in floating-point / Latency hardware

n \ k	4	8	16	32	64	128	256
4	5.6	4.9	4.8	4.6	4.5	4.4	4.2
8		3.9	3.2	3.0	2.7	2.6	2.5
16			2.3	1.8	1.7	1.6	1.7
32				1.5	1.2	1.5	1.6

Table 5.7. Ratio: Latency software in fixed-point / Latency hardware

The results are given in Tables 5.6 and 5.7.

The latency is overall larger with the software. For $n = 4$ and $n = 8$, the floating-point implementation should be preferred over the fixed-point implementation since it is both faster and more precise. For larger n it is the fixed-point implementation that is faster, the user can therefore choose either implementation depending on the specifications of the application and especially on how important the precision is compared to the speed.

With small n, the software is about 3 to 5 times slower than the hardware. For a same number k of samples per vector, a smaller number of vectors n means a higher number of instructions on small matrices, when the whole set of data is split up into $p = \lceil k/n \rceil$ square matrices of size $n \times n$. The instructions TMUL, ADD, and DIV are executed many times, which implies a lot of operations to decode the instructions, check the parameters, and initiate the computations. The overhead due to these additional operations is almost non-existent in the hardware accelerator, which explains the larger ratio of latency for smaller n.

For some combinations of n and k, the software is only 1 to 2 times slower than the hardware. This proves that the software implementation that will replace the hardware accelerator is a viable solution for some use cases regarding the performance, even though several DSPs will be needed to replace one accelerator while keeping the same throughput.

5.3 Power analysis

5.3.1 Toggle activity

The power dissipation in the DSP highly depends on the *toggle activity*. A *toggle* is a value change from 0 to 1 or from 1 to 0, and the corresponding activity is the number of toggle per clock cycle. This value is given as a percentage, and an activity of 100% means that the value changes at every cycle. The dynamic power dissipation in a component increases with this activity. The average *toggle activity* can be given for the whole DSP or for each of its sub-components.

It has been for instance measured for the first of the two matrix jobs described in the previous section, in which vectors are loaded and the corresponding covariance matrix is build. The simulations have been done with $n = 32$, $k = 256$, and internal floating-point format. The time window corresponding to the job execution has been divided into 600 slices, and the average *toggle activity* has been measured for each slice and for each sub-component of the DSP. The results are shown in Figure 5.9, for the whole DSP (first graph), for a sub-unit responsible for the register access and the arithmetic computations (second graph), and for a sub-unit responsible for fetching and decoding the instructions in the code (third graph).

5.3.2 Analysis

More than the values themselves, what can be examined in the results is the variations inside the time window. It can be noticed that the switching activity highly depends on the type of function that is executed. In the whole DSP, the same pattern is repeated four times: the activity is stable around 1.5%, then a plateau with roughly 5% of activity can be observed twice, followed by one or two peaks where the total average activity reaches almost 5.5%.

The low values at the beginning of the pattern correspond to the execution of 32 successive VLOAD matrix instructions. The assembly functions themselves are very short, most of the time is spent in the transitions between the functions. This code is therefore automatically compiled code, in which the number of DSP instructions per cycle is much lower than the theoretical maximum, which leads to a lower toggle activity. Then the TMUL instruction is executed twice. The total DSP activity reaches 5% for the assembly function, in which four arithmetic instructions are executed at most clock cycles. Then the outputs of the TMUL instructions are all added together with ADD instructions (only one call for the first iteration, two for the others to first add the two latest outputs of TMUL, and then add the result to the previous total). The duration of the ADD instruction is much smaller than for TMUL for $n = 32$. The ADD function is even more computationally intensive and uses more various registers, which explains that the activity is even higher during the ADD than during the TMUL assembly function.

These variations are not equally strong in all the sub-units inside the DSP. During the execution of the TMUL assembly functions, the average activity reaches

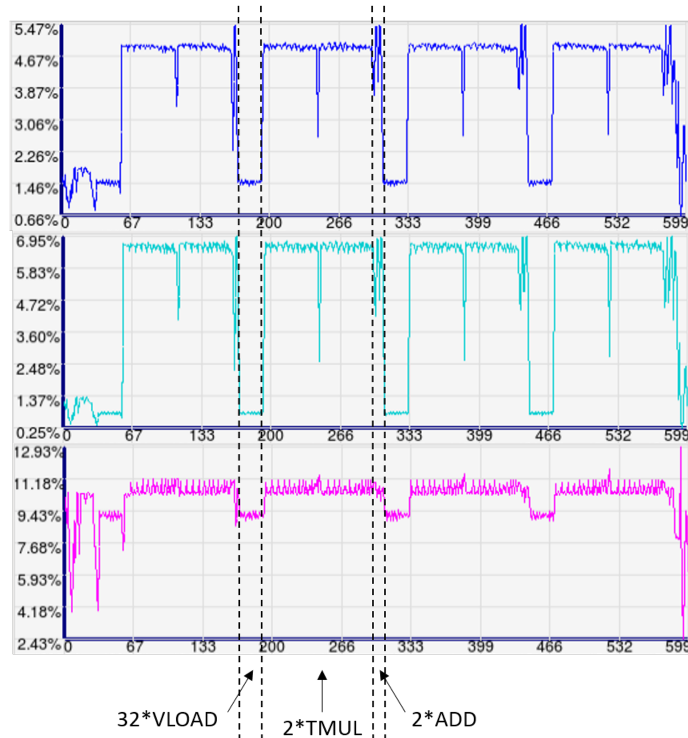

Figure 5.9. Average toggle activity during the execution of a matrix job, for the whole DSP (on the top), for a sub-unit responsible for the register access and the arithmetic computations (in the middle), and for a sub-unit responsible for fetching and decoding the instructions in the code (on the bottom). The time window is divided into 600 slices (horizontal axis).

6.5% in the block that contains all the computation units and the registers access, while it is under 1% during the transitional sections written in C. The difference is smaller for the instruction fetch unit which decodes the DSP instructions. This unit keeps a high activity during the whole execution, with an average of 9.5% during the VLOAD phases, and 11% for the arithmetic assembly functions. Indeed, this unit is always active whenever some code is executed.

5.3.3 Future work

Other tools can be used to measure the total power dissipation in the DSP. Information about the switching activity must be combined with data concerning the physical properties of the hardware in order to measure the dynamic power dissipation. The static power dissipation which is not affected by the toggle can also be measured. The same tests could be done for the hardware accelerator while it executes the same matrix job, to compare the cost of the software and hardware solutions.

It was not possible for practical reasons to make such a comparison during this project, mainly due to the tools accessibility and the long latency of the tests. Similar examples in the literature have proven that a software implementation of a program for a DSP architecture consumes 10 to 30 times more power than an equivalent hardware implementation (see the Background chapter). The programs which were implemented, and the properties of the hardware in the examples were however not the same as in this project. The clock frequency was for instance different in the software and in the hardware implementation in the examples from the literature.

With the architecture considered in this project, the DSP may have a higher activity especially because of the instruction fetch unit which is not needed in the accelerator, but for instance the activity linked to memory access is likely to be similar in both platforms. It can also be noticed that the average activity is probably higher in the computational units of the accelerator, since the transitions between the functions which use these units intensively are proportionally less important than in the software implementation.

It would be interesting to make some tests in the future to confirm or infirm these hypotheses. Even though the average power dissipation is expected to be higher for the software solution, and the energy consumption as well since this solution has a higher latency, some more precise comparison values would be useful for instance to make design decisions in the future. More precisely, they could be taken into account to decide if some other programs at Ericsson should be implemented in software or with a hardware accelerator.

6 Performance analysis

In this chapter, the speed of the software program is evaluated in more details for some instructions. The aim is first to identify and quantify the effects of the different bottlenecks which make the program slower. From this analysis, some evolutions that can lead to further optimisations in the system are suggested. These evolutions concern both the hardware (DSP instructions) and the software (client interface).

6.1 Identification of bottlenecks

It has been observed from the results of the previous chapter that for each matrix instruction, the experimental latency is significantly higher than the ideal latency, which is the time needed to perform only the useful arithmetic operations at the highest possible throughput. The aim now is to identify which are the steps in the code which are responsible for this additional latency.

6.1.1 Execution steps

When the program runs on the DSP, one or several operations are executed in each cycle. These operations can be classified in different execution steps that are described below.

For every matrix instruction, the first and the final steps are written in C language, while the main function is written in assembly. The main time-consuming steps in C that can be identified during the execution are the following:

- Opcode decoding: the opcode of the matrix instruction is compared with all the possible opcodes, and the right function is selected ("switch ... case" statement)
- Parameters check: the parameters of the instruction are checked and the execution is terminated if the combination of parameters is not valid
- Initialisation of assembly function: the accurate assembly function is selected given the information obtained from input matrices, and the matrix addresses are computed.

- Transition to next instruction: the information about the output matrix is updated, and the function returns

For the assembly functions, the following classification can be used to describe each cycle during the execution:

- Get function parameters: read each parameter of the function
- *Push*/*pop* context: *push* the current values of the registers in the stack before any other operation in order to save the context, and *pop* the values from the stack at the end of the function to restore the context
- *Branching*: jump in the code, conditionally or not, for instance to select the right implementation depending on whether the output must be transposed / conjugated
- Operation execution: call arithmetic operations or other type of operations
- Operation latency: wait for the result of the operations when the latency is greater than one cycle and no other operation can be executed (either because the result is needed immediately or because the register where the result is stored need to be reused for something else, typically in the next iteration of a loop)
- Other *move* instructions: read inputs, write outputs, update addresses, set number of iterations in a loop, etc.

An assembly function usually follows the structure described in Algorithm 6, with different levels of nested loops.

Algorithm 6 Example of structure for an assembly function

```
Get function parameters
Push context in the stack
Branch to another part of the code if transpose / conjugate output
Initialise values and addresses (move instructions)
loop
    Read inputs (move instructions)
    loop
        Execute arithmetic operations
    end loop
    Wait for end of arithmetic operations
    Write outputs and update addresses (move instructions)
end loop
Pop context from the stack
```

The latency of each step in C and in assembly is evaluated with the cycle accurate simulator. The proportion of each step in the total latency is then computed.

Note: Most *move* operations are run in parallel with other operations, and the cycles classified in the last assembly category "other *move* instructions" are the cycles in which only *move* instructions are called, and there is no need to wait for the result of some arithmetic operations (these cycles cannot be classified into any other category). Some of these cycles cannot be deleted even by optimising the schedule, especially when the algorithm contains nested loops. Indeed, those can be efficiently implemented with hardware loops in the DSP, in which all the branching operations are handled automatically without overhead, but in this case the schedule is constrained by the fact that two nested loops cannot begin or end at the same cycle. This introduces necessary transitional cycles, and the total experimental latency is therefore automatically increased compared to the ideal one.

Small errors of a few cycles can occur between the measured values and the exact values, especially for the steps written in C. The delimitation between each step is not as explicit in the compiled code as in the original C code, and the results are therefore obtained with small approximations.

Other variations can occur when conditional branching instructions are involved (instructions to jump to another part of the code only if a condition is verified). The branch prediction process in the DSP tends to minimise the latency of these instructions based on the previous evaluations of the conditions, so the latency depends on the succession of instructions that are executed.

Overall these errors and variations may only increase or decrease the total latency by a few cycles in the results below.

6.1.2 MUL in floating-point

The results obtained for MUL in floating-point are given in Table 6.1. The ideal latency for a matrix of size $n \times n$ is n^3, which corresponds to the execution of the multiply-accumulate operations (4 per cycle).

It can be noticed that this arithmetic step is fully optimised in the implementation, since its latency is exactly equal to the ideal latency (in bold in the table). For the larger values of n, this step is by far the longest step in the execution, with up to 97% of the total latency for $n = 32$. With $n = 4$ however, this step only lasts a quarter of the total execution time. In order to decrease the total latency significantly, it would be necessary to optimise the steps in the C code or the preliminary steps in assembly. This cannot be done easily since the latency of the C steps depend on the compilation process, and the first steps in the assembly function cannot be avoided.

6.1.3 MUL in fixed-point

In fixed-point, the value *lmax* must be computed for both inputs before actually calling the MUL assembly function. This value is computed with a function written in assembly that uses specific arithmetic operations. This additional function is

Steps	Latency (in cycles)	Proportion latency			
		$n = 4$	$n = 8$	$n = 16$	$n = 32$
Steps in C:					
- Decode opcode	37	15.5%	5.0%	0.8%	0.1%
- Check parameters	41	17.2%	5.6%	0.9%	0.1%
- Initialise assembly MUL function	23	9.7%	3.1%	0.5%	0.1%
- Transition to next instruction	17	7.1%	2.3%	0.4%	0.1%
Steps in the MUL assembly function:					
- Get function parameters	6	2.5%	0.8%	0.1%	0.0%
- *Push* / *pop* context	19	8.0%	2.6%	0.4%	0.1%
- *Branching*	6	2.5%	0.8%	0.1%	0.0%
- Arithmetic operation execution (*mac*)	$\boldsymbol{n^3}$	**26.9%**	**69.8%**	**91.3%**	**97.0%**
- Arithmetic operation latency (*mac*)	$\frac{1}{2}n^2$	3.4%	4.4%	2.9%	1.5%
- Other *move* instructions	$1 + 3n + \frac{1}{4}n^2$	7.1%	5.6%	2.5%	1.0%

Table 6.1. Detailed latency of the MUL instruction in floating-point for $n \times n$ matrices, and proportion of each step in the total latency for some values of n

called twice in the general case, only once if both inputs are equal (only the general case is considered here).

The ideal latency for a matrix of size $n \times n$ is in this case $\frac{1}{4}n^3$, which corresponds to the execution of 4 complex multiply-accumulate operations per cycle. With the results given in Table 6.2, it can be noticed that this step is again fully optimised, four complex multiply-accumulate operations are executed each cycle.

However, the proportion of this arithmetic step in the total latency is rather small, especially for the smallest values of n. The fixed-point implementation contains much more operations than the floating-point version: steps to compute the value *lmax* on both inputs, corresponding function call initialisations in the C code, and *shift* operations in the MUL assembly function (right shifts of output to avoid overflow). For $n = 4$ for instance, the optimised arithmetic step of the multiply-accumulate operations only take $\frac{1}{4}n^3 = 16$ cycles, which is very small especially compared to the constant cost of all the initialisation steps.

For larger values of n, the steps that are the most time-consuming with the main arithmetic step (*mac* operations) are the computation of *lmax* values, the time to wait for the result of the *mac* instructions, the *shift* operations, and, to a smaller

Steps	Latency (in cycles)	Proportion latency			
		$n = 4$	$n = 8$	$n = 16$	$n = 32$
Steps in C:					
- Decode opcode	37	10.8%	6.6%	2.0%	0.4%
- Check parameters	41	12.0%	7.3%	2.2%	0.4%
- Initialise assembly *lmax* function for input 1	35	10.2%	6.3%	1.9%	0.3%
- Initialise assembly *lmax* function for input 2	32	9.4%	5.7%	1.7%	0.3%
- Initialise assembly MUL function	24	7.0%	4.3%	1.3%	0.2%
- Transition to next instruction	23	6.7%	4.1%	1.2%	0.2%
Steps in *lmax* assembly function:					
- Get function parameters	2×4	2.3%	1.4%	0.4%	0.1%
- *Push* / *pop* context	2×17	9.9%	6.1%	1.8%	0.3%
- Arithmetic operation execution (compute *lmax*)	$2\times(\frac{1}{4}n+\frac{5}{16}n^2)$	3.5%	7.9%	9.1%	6.2%
- Other *move* instructions	$2 \times (4 + \frac{1}{4}n)$	2.9%	2.1%	0.9%	0.2%
Steps in the MUL assembly function:					
- Get function parameters	7	2.0%	1.3%	0.4%	0.1%
- *Push* / *pop* context	23	6.7%	4.1%	1.2%	0.2%
- *Branching*	7	2.0%	1.3%	0.4%	0.1%
- Arithmetic operation execution (*mac*)	$\boldsymbol{\frac{1}{4}n^3}$	**4.7%**	**22.9%**	**55.4%**	**77.9%**
- Arithmetic operation latency (*mac*)	$\frac{1}{2}n^2$	2.3%	5.7%	6.9%	4.9%
- Arithmetic operation execution (*shift* on result)	$\frac{1}{2}n^2$	2.3%	5.7%	6.9%	4.9%
- Other *move* instructions	$1 + 3n + \frac{1}{4}n^2$	5.0%	7.3%	6.1%	3.4%

Table 6.2. Detailed latency of the MUL instruction in fixed-point for $n \times n$ matrices, and proportion of each step in the total latency for some values of n

extent, the *move* instructions.

6.1.4 Other instructions with arithmetic computations

In the other instructions that contain arithmetic operations, the bottlenecks are often the same as for MUL.

For small values of n, an important part of the execution time is spent in the C code, and for the initialisation steps in assembly. All these parts cannot be easily optimised.

For larger n, the throughput of the arithmetic operations become the main bottleneck. For the instructions that involve divisions and/or square-roots like CHOL and DIV, the latency of the arithmetic instructions also increases the total duration of the function. These operations have indeed a long latency, and it is not always possible to compute the divisions and square-roots in advance in the schedule.

6.1.5 VLOAD

The VLOAD instructions has also been analysed in more details. In this instruction, a vector is loaded in a row that can cover several matrices in *smx_mem* (see Appendix A).

When data is loaded directly from the floating-point single-precision complex format to the same format, the assembly function contains only initialisation steps and a few load requests sent to the shared memory. In the C code, the most time-consuming step is the "Transition to next instruction" part, in which information about the output matrices is updated in *smx_mem_manager* (erase all the previous information and write internal format). If the length is large, especially when it is large compared to the final matrix size n, a lot of changes must be made in *smx_mem_manager*. For $n = 4$ and a vector length equal to 256 for instance, the vector covers $256/4 = 64$ different matrices. In this case, this last C steps takes almost 80% of the total execution time of the VLOAD function, so this is by far the main bottleneck of the function.

The VLOAD instruction can support various external formats. When a conversion is needed during the VLOAD instruction, data is first loaded in an intermediate buffer, and then converted and copied into *smx_mem*. This conversion step can also be a bottleneck of the function. An external format that is often used for matrix algorithm at Ericsson is the complex $Q5.3$ fixed-point format. When a vector of 256 coefficients is loaded into a succession of 4×4 matrices and converted in floating-point format, the conversion step takes approximately 20% of the total latency of VLOAD. The C step to update output information takes in this case roughly 60% of the total time. When the external format is the complex fixed-point $Q1.15$ format, the conversion takes about 15% of the total time.

6.2 Hardware enhancements

Suggested evolutions in the DSP instruction set could increase the speed of some assembly functions.

6.2.1 Possible optimisations in the *shift* instruction

For the functions that mostly contain arithmetic operations like MUL, the main bottleneck is the throughput and latency of the multiply-accumulate instruction, which cannot be changed easily. Other optimisations can be possible however for other operations, like the *shift* instructions in the fixed-point implementation of MUL.

In the current implementation of MUL to compute $C = AB$, four consecutive coefficients $c_{(i)(4j...4j+3)}$ are computed simultaneously. After a loop in k, the values of the real parts and imaginary parts of $c_{(i)(4j...4j+3)} = \sum_{k=0}^{n-1} a_{(i)(k)} b_{(k)(4j...4j+3)}$ are each stored in 40-bit registers. The results then need to be right-shifted by a given number of bits, in order to avoid overflow. The data from bit 16 to bit 31 in all the destination registers is finally written in memory.

Current *shift* instruction

The current *shift* instruction that is used here to perform the arithmetic shifts has the following parameters:

- 40-bit source register that contains the input data to shift
- 16-bit source register that contains the number of bit shifts
- 40-bit destination register that will contain the output

Four instructions of this type can run in parallel at every cycle and the latency is equal to one cycle (result available immediately next cycle after the instruction is called). Because it operates on a 40-bit input, the *shift* instruction must be called for each of the real and imaginary parts of the coefficients, as shown in Figure 6.1.

Even though steps (1) and (2) in Figure 6.1 can be executed in parallel in the same cycle, the *shift* instruction must be called twice for each coefficient. In the MUL function, where four coefficients must be shifted at the same time and then written in *smx_mem*, the instructions are currently scheduled in the following way:

Cycle 1	*Shift* $c_{(i)(4j)}.re$	*Shift* $c_{(i)(4j)}.im$
	Shift $c_{(i)(4j+1)}.re$	*Shift* $c_{(i)(4j+1)}.im$
Cycle 2	*Shift* $c_{(i)(4j+2)}.re$	*Shift* $c_{(i)(4j+2)}.im$
	Shift $c_{(i)(4j+3)}.re$	*Shift* $c_{(i)(4j+3)}.im$
	Write $c_{(i)(4j...4j+1)}$	
Cycle 3	*Write* $c_{(i)(4j+2...4j+3)}$	

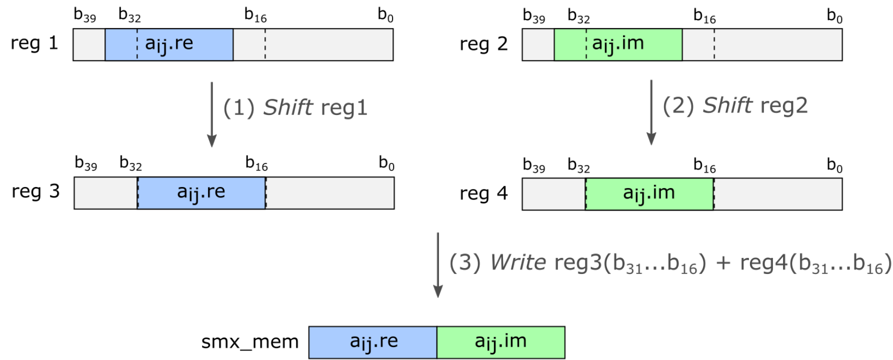

Figure 6.1. Current *shift* instruction

New suggested instruction

In order to increase the speed of this step, it would be necessary to increase the throughput of the *shift* instruction, so that it can operate on 40-bit register pairs, instead of simple registers.

In Figure 6.1 this would mean that steps (1) and (2) are merged into a single instruction, "*Shift* reg1 + reg2", that would result in the output register pair, reg3 + reg4. If the maximum number of instructions per cycle and the latency of the instruction are not changed, the schedule for MUL could become the following:

Cycle 1 *Shift* $c_{(i)(4j)}$ *Shift* $c_{(i)(4j+1)}$ *Shift* $c_{(i)(4j+2)}$ *Shift* $c_{(i)(4j+3)}.im$

Cycle 2 *Write* $c_{(i)(4j...4j+3)}$

However, this schedule cannot be implemented in practice because of the *Write* step in cycle 2. If the structure of the *shift* instruction is not modified further, both the input and output data will be stored in 40-bit register pairs. This would mean that during cycle 2, the data for the four coefficients must be read from 8 different registers.

In the current schedule of the MUL instruction, other *move* instructions must also be executed at the same time, typically to read inputs for the next iteration of the loop in which $c_{(i)(4j+4...4j+7)}$ will be computed. Because of the limitations in the number and combination of *move* operations in the same cycle, the last *Write* step would actually take two cycles instead of one. As a result, there would be no gain in speed with the new *shift* instruction.

A solution could be to modify again the specification of the *shift* instruction. Instead of returning the output in a pair of 40-bit registers, the useful data (data from bit 16 to 31) from both these registers would be packed into one final register. If $r1$ and $r2$ are the input registers, $r3$ is the output register, ls is the number of left shifts to be performed (can be negative and in this case represents the number

of right shifts), and $temp1$ and $temp2$ are two temporary variables, the following operations are executed ($rk[b_j \dots b_i]$ refers to the data contained in rk between bit i and bit j both included):

$$
\begin{aligned}
temp1 &= r1 << ls \\
temp2 &= r2 << ls \\
r3 &= 0 \\
r3[b_{31} \dots b_{16}] &= temp1[b_{31} \dots b_{16}] \\
r3[b_{15} \dots b_0] &= temp2[b_{31} \dots b_{16}]
\end{aligned}
$$

This *shift* instruction could be instead renamed *extpack* for “extract-and-pack”. The parameters are the following:

- pair of 40-bit source registers that contains the input data to shift ($r1$ and $r2$)
- 16-bit source register that contains the number of bit shifts (ls, same for both input registers)
- 40-bit destination register that will contain the output ($r3$)

The operations that are performed can be written in the following simpler way (see Figure 6.2):

$$
\begin{aligned}
r3 &= 0 \\
r3[b_{31} \dots b_{16}] &= r1[b_{31-ls} \dots b_{16-ls}] \\
r3[b_{15} \dots b_0] &= r2[b_{31-ls} \dots b_{16-ls}]
\end{aligned}
$$

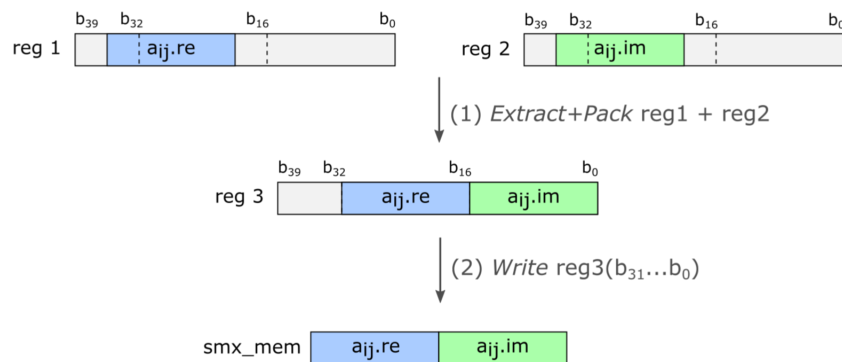

Figure 6.2. Extract and pack (*extpack*) suggested new instruction

This new instruction can only lead to a gain of time if its latency is equal to one cycle, and if four instructions of this type can run in parallel.

Concerning the MUL instruction, the new latency that would be obtained after these changes has been evaluated. The corresponding gain in time has also been computed with the formula (*old_latency* - *new_latency*) / *old_latency*. The gain would be of 1.2% for $n = 4$, 2.9% for $n = 8$, 3.5% with $n = 16$, and 2.4% for $n = 32$.

The gain in time for the whole function is limited because the time spent to execute the *shift* instructions represented a small proportion of the whole latency from the beginning. However, none of the other steps in the assembly function can easily be optimised with small changes in the hardware. Furthermore, this new instruction could be used for other software programs. Every time a programmer wants to accumulate product of complex $Q1.15$ fixed-point inputs in pairs of 40-bit registers ($Q8.30$ format for real and imaginary parts), the instruction can be used to convert the output back in the complex $Q1.15$ format, after shifting the results.

Additional options

The performance could be improved further if a few additional options are added to the new extract-and-pack instruction. These options can be implemented either as different instructions, or by adding a configuration parameter in the initial instruction, that would enable the programmer to set or unset each option.

First, an option could be added to conjugate the output of the operation. Even though it may not always be the case, the instruction is mostly meant to be used for complex numbers. Sometimes it may be needed to conjugate the data before writing it in *smx_mem*, as for instance in the MUL_C function. The schedule to finalise the result in this function was initially the one below:

Cycle 1	*Negate* $c_{(i)(4j)}.im$	*Negate* $c_{(i)(4j+1)}.im$
	Negate $c_{(i)(4j+2)}.im$	*Negate* $c_{(i)(4j+3)}.im$
Cycle 2	*Shift* $c_{(i)(4j)}.re$	*Shift* $c_{(i)(4j)}.im$
	Shift $c_{(i)(4j+1)}.re$	*Shift* $c_{(i)(4j+1)}.im$
Cycle 3	*Shift* $c_{(i)(4j+2)}.re$	*Shift* $c_{(i)(4j+2)}.im$
	Shift $c_{(i)(4j+3)}.re$	*Shift* $c_{(i)(4j+3)}.im$
	Write $c_{(i)(4j...4j+1)}$	
Cycle 4	*Write* $c_{(i)(4j+2...4j+3)}$	

The 4 cycles could already be reduced to 3 cycles with the standard *extpack* instruction (cycles 2 and 3 can be merged), but an additional cycle could be gained with an “extract-and-pack-and-conjugate” instruction. It would replace all the operations in the first three cycles of the schedule.

This option would also be beneficial for the TMUL function. Because the output matrix is Hermitian, every time a coefficient c_{ij} with $j > i$ is computed, the coefficient $c_{ji} = \overline{c_{ij}}$ can directly be deduced. The simple *extpack* instruction would not result in a gain of time in TMUL, but the latency would slightly decrease if

an additional conjugate option is available in the instruction. The gain for TMUL would be approximately between 1.3% and 3.5%. It is a necessary condition that the latency and throughput of the instruction remain the same with the option.

This possibility in an instruction to obtain a conjugate result more quickly does not have an equivalent in floating-point: because only real numbers are manipulated in this format, the sequence of real operations in a complex multiplication can simply be modified to obtain the conjugate result with no overhead. When both a complex number and its conjugate are needed like in TMUL, the operation to negate the imaginary part cannot be avoided, and cannot be done at the same time as another other useful operation like it could be done with a *shift* in fixed-point.

Another possible option would be to enable a correct rounding in the result. When the output coefficients are accumulated in 40-bit registers, a lot of significant bits are used to maintain the precision. But then only the 16 useful bits are extracted, which is equivalent to a truncation in the result. In order to have a correct rounding, the number $(1 << 15 - ls)$ can be added to the inputs in $r1$ and $r2$ before extracting the result (in other words 2^{-16} is added to a number which is then truncated to $Q1.15$ format with 15 significant bits). Of course this option would only have influence on the precision of the result, not on the speed.

With this option, the error on the output of the MUL function for the inputs generated in the previous chapter would decrease by approximately 23%. This option could be implemented for the *extpack* instruction, or for the initial *shift* instruction, if the extract-and-pack instruction is finally not implemented.

6.2.2 Load/store instructions with on-the-fly conversion

Another way to improve the performance of the system would be to optimise the load/store transfers. A common use case is when a vector is loaded from an external format to the complex floating-point format. It has been shown that the conversion step, if it is needed, can be time-consuming.

For a transfer of 256 coefficients in 4×4 matrices, the conversion step takes 20% of the total time for $Q5.3$ external format, and 15% of the total time for $Q1.15$ external format. If only the assembly VLOAD function is considered without the C code, this proportion gets much more important, since most of the time in this function is spent for the conversion. The same can be observed for the equivalent VSTORE function.

A suggestion of new DSP instruction would be a set of "load+convert" and "convert+store" instructions, in which data is loaded or stored, and converted before being written at the destination address. With these instructions, the VLOAD and VSTORE instruction could be implemented simply by sending a few load/store request even when a conversion must be done. There would be no need to use an intermediate buffer and to do a manual conversion.

This is only possible when the internal format is floating-point. When internal fixed-point format is used, the data size is smaller, and the matrix blocks are

therefore not stored consecutively in memory. The use of an intermediate buffer is therefore necessary in this case.

The following set of instructions could be implemented:

- Load from shared memory + convert from complex $Q1.15$ fixed-point format to complex floating-point format
- Load from shared memory + convert from complex $Q1.7$ fixed-point format to complex floating-point format
- Convert from complex floating-point format to complex $Q1.15$ fixed-point format + store in shared memory
- Convert from complex floating-point format to complex $Q1.7$ fixed-point format + store in shared memory

All these instructions would take as parameters a source address, a destination address and a size. A scaling factor could also be specified, that would enable for instance to use easily the $Q5.3$ format or any other fixed point 8-bit and 16-bit formats.

If only the VLOAD assembly function is considered, the gain in time would be very high. For smaller lengths, the gain is around 20%. For a length equal to 256, the gain reaches 65% with a conversion from $Q1.15$ format, and 75% from $Q1.7$ (or $Q5.3$) format.

These values have been computed theoretically in the ideal case when there is no additional delay to access data in the shared memory. In the worst case when too many load/store requests are in progress and no new request can be sent before the first ones have been completed, the gain in time generated by the new instruction still remain over 15% for $Q1.15$ external format, and above 30% for $Q1.7$ format.

Similar values have been obtained for the assembly VSTORE function. Concerning the whole function including the C code, the gain in time depends on the matrix size n, but it is in general much lower in the current implementation.

More generally, these new instructions could be useful for all the LOAD, STORE, VLOAD, VSTORE functions even when an intermediate buffer is used and data must be manually copied from/to the buffer to/from *smx_mem*. The main advantage is a significant simplification of the code. If all the external and internal formats are considered, in addition all to the possible transformations (transpose and/or conjugate), the code would be very complex with a different implementation for each case. If at least the conversion is done inside the load/store instructions, the number of different copy functions between the buffer and *smx_mem* is much reduced. This makes the code simpler and more easily maintainable.

Two additional instructions could be added to further simplify the code when the internal format is $Q1.15$ fixed-point format:

- Load from shared memory + convert from complex $Q1.7$ fixed-point format to complex $Q1.15$ fixed-point format
- Convert from complex $Q1.15$ fixed-point format to complex $Q1.7$ fixed-point format + store in shared memory

These last conversions are easy to implement since they only increase or decrease the number of significant bits. A scaling factor can also be used during the conversion.

In some cases, the on-the-fly conversion also leads to an increased speed, especially for the conversion between $Q1.7$ format and the floating-point format, which is time-consuming if done manually (need to convert between $Q1.7$ and $Q1.15$ format, and between $Q1.15$ and floating-point format).

If other common use cases which involve other type of conversions are identified in the future, the corresponding instructions can be added in the instruction set. A conjugate option could for instance be added in the load/store instruction, to negate one coefficient out of two in a given format. Conversion between external real formats and internal complex formats can also be made easier, by simply filling one coefficient out of two with 0s in the case of a load, or by deleting one coefficient out of two for the equivalent store instruction.

The speed of the VLOAD and VSTORE functions could be increased with these instructions, and this would also lead to further simplifications in the code.

6.3 Suggested changes in the client interface

The specifications for the client interface, and especially the set of matrix functions, have been chosen for the hardware accelerator when it was designed. The choices have been made based on the programmers' need, but also based on the characteristics of the hardware accelerator.

When the same program is implemented in software for Ericsson's DSP core, a different interface with the client could lead to better performance, since the characteristics of the hardware differs from the accelerator.

6.3.1 Revised instruction set

The first main difference between the DSP and the accelerator is that the accelerator has a better ability to, first, read and write coefficients row-wise and column-wise with the same complexity, and secondly, to conjugate output coefficients without adding any latency.

As a result, most matrix functions can be implemented easily with a transpose and/or conjugate version, that have the same latency as the standard function. With the DSP however, this is not always possible.

The ADD instruction for instance computes the output X_0 with the formula $X_0 = X_0 + \lambda X_1$. This can easily be computed with two nested loops for the rows and columns.

In order to execute the ADD_T function to compute $X_0 = (X_0 + \lambda X_1)^T$ while doing all the computations in place, it would be necessary for each output coefficient $(x_0)_{ij}$ with $j > i$ to compute at the same time the coefficient $(x_0)_{ji}$. Compared to the simple ADD instruction, this would involve more pointers, and a more complex schedule.

For the ADD_C instruction in which $X_0 = \overline{X_0 + \lambda X_1}$ is computed, more operations would be needed to conjugate every coefficient. As a consequence, the latency of this function would be much higher than the latency for the simple ADD function. The same can be said about ADD_TC instruction.

In order to simplify the code, all these instructions that cannot be implemented efficiently could be removed from the instruction set. This concerns all the _T, _C, and _TC versions of the instructions SCALE, ADD, VMUL, ABS2, VINV, and VSQRI. The CONST_C instructions could also be deleted, since it is equivalent to CONST with a conjugated λ parameter.

The transpose and conjugate options could therefore only be kept for MUL, TMUL, and DOT, since they do not affect the latency or only in a not significant way. Because the output of the TMUL function is Hermitian, the instructions TMUL_T (equivalent to TMUL_C) and TMUL_TC (equivalent to TMUL) could be deleted.

New instructions could be added to simply transpose, conjugate, or do both operations on a matrix, in place or with a copy.

The aim of these changes is to provide the programmer with a set of instructions that can be implemented efficiently, and it is then the role of the programmer to identify the best succession of instructions for a given matrix algorithm. The set of available instructions can change over time, if new use cases are identified. It could be for instance possible in the future to introduce instructions that would take transposed inputs as parameters, if these functions could be implemented efficiently.

For example, if a user wants to compute the matrix $C = (C + D)^*$, and then $A = BC$, it may be more efficient to compute instead $C = C + D$, and then $A = BC^*$. If this kind of features could be useful, a new modified MUL instruction with parameters X_0, X_1, and X_2 could be introduced to compute $X_0 = X_1 X_2^*$.

Another new feature that could be useful concerns the DIV instruction, in which $X_0 = X_1^{-1} X_0$ is computed. Sometimes it can be useful to compute simply the inverse X_1^{-1} of a matrix. For the moment the only way to do this is to load first an identity matrix into X_0, and then to call DIV instruction. The LOAD increases the latency of the operation, especially if the matrix size is high, and it also implies that an identity matrix must be stored in the shared memory.

A possible solution would be to add an option in the DIV instruction to compute

n \ k	4	8	16	32	64	128	256
4	2.6	2.3	2.2	2.2	2.1	2.2	2.2
8		2.0	1.8	1.8	1.8	1.9	2.0
16			1.6	1.5	1.6	1.7	1.9
32				1.6	1.5	1.8	2.0

Table 6.3. Ratio: Latency software in floating-point / Latency hardware

a simple inverse, in which an identity matrix would be first written in X_0 before beginning the division.

6.3.2 Matrix job templates

Another bottleneck that has been identified in most instruction is the step to decode the opcode and to check the parameters. Other steps written in C can also be time-consuming, like the steps in VLOAD to update the information in *smx_mem_manager* about all the matrices covered by the loaded vector. If a matrix job is repetitively sent to a Matrix server, with the same succession of instructions and parameters, and only different input and output addresses in the shared memory, all these time-consuming steps in C could be avoided.

A desirable feature in the matrix program would therefore be to provide a set of job templates for the most common use cases. Instead of building a matrix job as a list of matrix instructions, the client would simply send a request to execute a job template. Only the input and output addresses would be specified, and maybe also some other parameters like the matrices' size.

These jobs would be implemented simply in the Matrix server DSP with a succession of function calls to the optimised assembly functions, and almost no intermediate steps in C (only need to compute the internal addresses of the matrices).

This has been tested with the two matrix jobs studied in the previous chapter. Both external and internal formats are complex floating-point format. The ratio of the total latency obtained with the new software version over the total latency obtained with the hardware accelerator has been evaluated. The results are given in Table 6.3.

Compared to what had been obtained with a matrix job built with matrix instructions, the speed of the software has been globally increased, especially for $n = 4$ and $n = 8$. The software is now only up to 2.6 times slower than the hardware, for a similar precision in the results. For $k = 256$, the software is not much more than 2 times slower than the hardware. As a comparison, the ratio obtained for $n = 4$ and $k = 256$ with a standard matrix job was 3.7.

The performance could be improved further by slightly modifying the implementation of the matrix functions and the data structures. Some functions like TMUL and the forward- or backward-substitution in DIV could be implemented for rectangular matrices instead of square matrices with only little modifications in the code. The data structure for matrix storage could also be modified for this use case, in order to be able to store full vectors with consecutive coefficients in memory. In the first of the two matrix jobs considered here for instance, the covariance matrix could be built with only one call to TMUL on a large rectangular matrix instead of multiple calls to TMUL and ADD functions on square matrices.

Another advantage of the job templates is that the client would not need to build the matrix instructions one by one and to store them in the shared memory before sending a request to the server, only a few parameters would now need to be set. Since the job templates are written in software, they can also easily be updated if the needs of the programmers change over time.

Furthermore, the job templates could even be implemented as a function that could be executed directly by the client. Now that the building blocks of matrix algorithms have been implemented and optimised in software, the code can also be directly used by the client for some algorithms.

7 Conclusion

The aim of this project was to design and to optimise a linear algebra software program for Ericsson's DSP core. This program can execute matrix jobs, composed of a succession of functions applied on matrices. This software is intended to replace an existing hardware accelerator, that currently implements the same functionality. A software solution is indeed more flexible and can be easily adapted to meet the changing requirements of the users. It can also lead to a more optimised use of resources on the chip, since a DSP can execute various kind of applications when it does not run the matrix program, while this was not possible with a specific-purpose accelerator. This can compensate for the expected lower performance of the software implementation regarding speed and power dissipation.

In a preliminary theoretical analysis, the possible algorithms, data formats and data structures have been identified and compared in terms of speed and precision, given the hardware characteristics of the DSP. From the results of this analysis, several versions of each matrix function have been implemented. During the execution of the program, the best version is chosen depending on the current state of the matrices for the algorithms, and the preferences of the user for the data formats.

The program has been implemented for the upcoming version of the DSP. The main structure has been written in the C language, and the matrix functions themselves have been written directly in assembly. It has been proven in this project that the software implementation is a viable solution with reasonable performance when compared with the hardware accelerator.

The choice on internal data format can be made between a floating-point and a fixed-point format.

The first one offers the same precision as in the hardware accelerator, but a lower speed, since the maximum throughput of the computational units in the DSP is four times slower than in the accelerator for most arithmetic operations. The tests have proven that the software is in practice slower than the hardware, but in many cases less than four times slower. This has been achieved by choosing accurately the algorithms and by optimising the schedule of the instructions in the assembly code to make the most of the DSP's capabilities.

Another possibility that exists only in the software is to use fixed-point internal computations. That leads to less precise results, but better speed performance. The software can be in some cases almost as fast as the hardware, but the speed is

usually limited by all the necessary operations to manage data range and to avoid overflows in this format. It can be yet an interesting alternative option for some users, when it is not necessary to obtain results with a high precision.

The performance of the functions could be increased further by adding some new instructions in the DSP that would make some of the execution steps a bit faster. However, the main limitations of the functions concerning speed are the throughput of the main arithmetic instructions in the DSP, and the code that is executed between every assembly function and which cannot be fully optimised in C.

While the first bottleneck cannot change easily without making important modifications in the hardware, the second one could be reduced by making some changes in the program's specifications and the client interface. The creation of job templates could for instance make the program faster since the succession of matrix functions and parameters is known in advance. A test has proven that the software could be only 1.5 to 2.5 slower in this configuration to execute a matrix algorithm used at Ericsson.

The flexibility of the software is probably the main potential source for further improvements in the program. Depending on the needs of the programmers that can change over time, the matrix instruction set and the actual implementation can be modified as often as needed to obtain the best possible performance.

A current limitation of the performance for complex matrix algorithm is that it is not always optimal to divide them into a succession of functions applied on square matrices stored in the currently used data structure. What could be done is to add small modifications in the assembly functions in order to make them more modular, and especially to support more matrix shapes and storage structures. The assembly functions could then also be used directly by the client like library functions to build optimised algorithm, in addition to being used in the current matrix program.

Another possible continuation of this project concerns the power dissipation. Measuring precisely the dissipation in the DSP with the matrix program and comparing the results with the power dissipation of the existing hardware accelerator would complete the comparison analysis between the two solutions. This could be taken into account together with the rest of the performance analysis to decide if future programs should be implemented in software or with a specialised accelerator. The dynamic power dissipation in the DSP could also be analysed further, in order to understand if some simple changes in the code can lead to a less power consuming program.

Bibliography

[1] A. Fehske, G. Fettweis, J. Malmodin, and G. Biczok. The global footprint of mobile communications: The ecological and economic perspective. *IEEE Communications Magazine*, 49(8):55–62, August 2011.

[2] Institute of Electrical and Inc. Electronics Engineers. IEEE Code of Ethics.

[3] R. Vejanovski, J. Singh, and M. Faulkner. ASIC and DSP implementation of channel filter for 3g wireless TDD system. In *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 47–51, 2001.

[4] H. K. Boyapati and R. V. Raja Kumar. A comparison of DSP, ASIC, and RISC DSP based implementations of multiple access in LTE. In *2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP)*, pages 1–5, March 2010.

[5] Texas Instrument. Choosing the Right Architecture for Real-Time Signal Processing Designs - White Paper, November 2002.

[6] Institute of Electrical and Inc. Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[8] N. Higham. *Accuracy and Stability of Numerical Algorithms*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, January 2002.

[9] G. Peters and J. H. Wilkinson. On the Stability of Gauss-Jordan Elimination with Pivoting. *Commun. ACM*, 18(1):20–24, January 1975.

[10] C. Inacio and D. Ombres. The DSP decision: fixed point or floating? *IEEE Spectrum*, 33(9):72–74, September 1996.

[11] Wayne T. Padgett and David V. Anderson. Fixed-Point Signal Processing. *Synthesis Lectures on Signal Processing*, 4(1):1–133, January 2009.

[12] Zoran Nikolić, Ha Thai Nguyen, and Gene Frantz. Design and Implementation of Numerical Linear Algebra Algorithms on Fixed Point DSPs. *EURASIP Journal on Advances in Signal Processing*, 2007(1):087046, 2007.

[13] J. Janhunen, T. Pitkanen, O. Silven, and M. Juntti. Fixed- and Floating-Point Processor Comparison for MIMO-OFDM Detector. *IEEE Journal of Selected Topics in Signal Processing*, 5(8):1588–1598, December 2011.

[14] James H. Wilkinson. *Rounding Errors in Algebraic Processes.* Her Majesty's Stationery Office, London, 1963.

[15] M. Martel, A. Najahi, and G. Revy. Toward the synthesis of fixed-point code for matrix inversion based on Cholesky decomposition. In *2014 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–8, October 2014.

Appendix A

Set of matrix instructions

A defined set of matrix instructions is supported by the existing hardware accelerator. Every instruction contains parameters, among which an operation code (*opcode*), the size of the matrices, and operands for the input/output matrices. The size of the matrices is always a multiple of 4.

The instruction set can be divided into two groups:

- Instructions to load/store data from/to the shared memory to/from the internal memory of the accelerator;
- Instructions to execute operations on matrices which are stored internally;

For most instructions, it is also possible to directly transpose and/or conjugate the output of the function. If INSTR is the name of a simple instruction, INSTR_T is the name of the instruction that does the same operation and return a transposed output. Similarly, INSTR_C defines the same instructions with a conjugated output, and with INSTR_TC the output is both transposed and conjugated Example:

MUL $\leftrightarrow X_0 = X_1 X_2$

MUL_T $\leftrightarrow X_0 = (X_1 X_2)^T$

MUL_C $\leftrightarrow X_0 = \overline{X_1 X_2}$

MUL_TC $\leftrightarrow X_0 = (X_1 X_2)^*$

Every matrix that is stored in the internal memory is uniquely identified by its size n and its operand. The internal memory is a $N \times N$ table that is divided into blocks of size $n \times n$, as shown in Figure A.1, and each operand for this size n refers to one of the blocks.

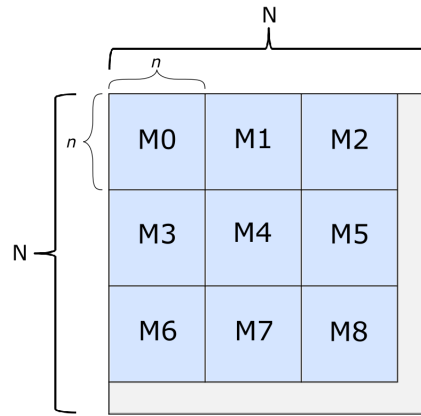

Figure A.1. Location of the matrices of size $n \times n$ in the internal memory table of size $N \times N$

A.1 Load - Store instructions

The set of instructions for data transfer is described in Table A.1 and examples are displayed in Figure A.2.

In the shared memory, the transferred data is always stored consecutively from a specified address. For each instruction, many different external formats are supported.

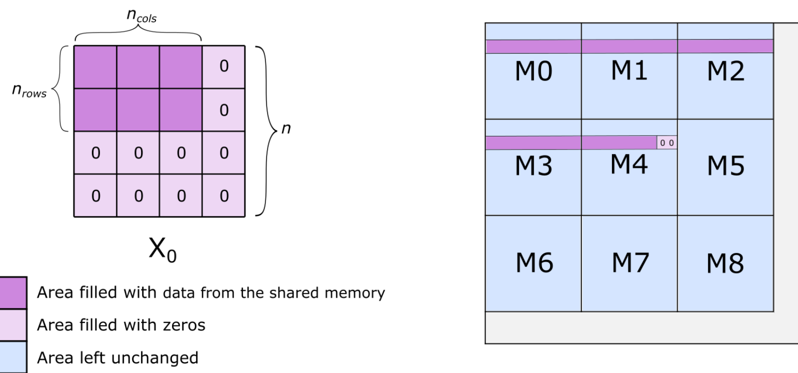

Figure A.2. Example of LOAD (on the left) and VLOAD (on the right) instructions

Instructions	Description
LOAD LOAD_T LOAD_C LOAD_TC	Load n_{rows} consecutive rows of n_{cols} columns from the shared memory to the internal memory. The destination address is given by the operand X_0. If $n_{rows} < n$ and/or $n_{cols} < n$, the remaining coefficients are filled with zeros (see Figure A.2). Matrices are if requested transposed/conjugated before being written in the internal memory.
STORE STORE_T STORE_C STORE_TC	Store n_{rows} rows of n_{cols} columns from matrix X_0 in the internal memory to the shared memory. With transpose/conjugate options, only the output is modified, the internal matrix remains the same.
VLOAD VLOAD_T VLOAD_C VLOAD_TC	Load a vector with a defined *length* from the shared memory to the internal memory. Only the *row* number can be specified in the destination address, the data is then written in this row in the first block (M0 in Figure A.2). If *length* $> n$, the data is then written successively in M1, M2, M3, etc. If *length* is not a multiple of n, the last coefficients of the last block are filled with zeros. In case of transposition, the data is written in a column instead of a row.
VSTORE VSTORE_T VSTORE_C VSTORE_TC	Store a vector with a defined *length* from the internal memory to the shared memory. The source data is read from a specified *row* in the successive blocks M0, M1, M2, M3, etc. For VSTORE_T and VSTORE_TC, data is read from a column instead of a row.

Table A.1. Instruction to load/store data

A.2 Other instructions

For every other instruction, the definition is given for the simple instructions INSTR, the output for the instructions INSTR_T, INSTR_C, INSTR_TC can easily be deduced by transposing and conjugating the output X_0 of the operation. The description of these operations can be found in Table A.2.

Instructions	**Description**
CONST CONST_C	X_0 is filled with a constant λ
SCALE SCALE_T SCALE_C SCALE_TC	$X_0 = \lambda X_1$
ADD ADD_T ADD_C ADD_TC	$X_0 = X_0 + \lambda X_1$
MUL MUL_T MUL_C MUL_TC	$X_0 = X_1 X_2$
TMUL TMUL_T TMUL_C TMUL_TC	$X_0 = X_1 X_1^*$
DOT DOT_T DOT_C DOT_TC	First row of X_0 = Diagonal of $X_1^* X_2$. Other rows in X_0 are filled with zeros.
DIV	$X_0 = X_1^{-1} X_0$
CHOL	In-place Cholesky decomposition. Output X_0 is equal to the lower-triangular matrix L where input X_0 satisfies $X_0 = LL^*$
LDL	In-place LDL decomposition. Output X_0 is equal to the lower-triangular matrix L with ones on the diagonal, and output X_1 is equal to the diagonal matrix D where the input X_0 satisfies $X_0 = LDL^*$
VMUL VMUL_T VMUL_C VMUL_TC	$\forall i, j \in \{0 \ldots n-1\}, (x_0)_{ij} = (x_1)_{ij}(x_2)_{ij}$ (Elementwise multiplication)
ABS2 ABS2_T	$\forall i, j \in \{0 \ldots n-1\}, (x_0)_{ij} = \|(x_1)_{ij}\|^2$ (Elementwise square absolute value)
VINV VINV_T	$\forall i, j \in \{0 \ldots n-1\}, (x_0)_{ij} = 1/(x_1)_{ij}.re$ (Elementwise inversion)
VSQRI VSQRI_T	$\forall i, j \in \{0 \ldots n-1\}, (x_0)_{ij} = 1/\sqrt{(x_1)_{ij}.re}$ (Elementwise square root inversion)
DMAX	$X_0 = mI_d$ where $m = \max(\{\|(x_1)_{ii}\| : 0 \leq i \leq n-1\})$ and I_d is the identity matrix

Table A.2. Other instructions

Appendix B

Division algorithm

Compute $A^{-1}B$, with $A, B \in \mathbb{C}^{n \times n}$ Sources: [7] [8]

Algorithm 7 Gaussian elimination with partial pivoting

for $k = 0 \ldots n-1$ **do**
- ▷ STEP 1: search pivot
- $max_pivot \leftarrow |a_{kk}|^2$
- $max_pivot_row \leftarrow k$
- **for** $i = k+1 \ldots n-1$ **do**
  - Compute $|a_{ik}|^2$
  - **if** $|a_{ik}|^2 > |a_{kk}|^2$ **then**
    - $max_pivot \leftarrow |a_{ik}|^2$
    - $max_pivot_row \leftarrow i$
- ▷ STEP 2: switch rows
- **if** $max_pivot_row > k$ **then**
  - **for** $j = k \ldots n-1$ **do**
    - $a_{kj} \leftrightarrow a_{ij}$
  - **for** $j = 0 \ldots n-1$ **do**
    - $b_{kj} \leftrightarrow b_{ij}$
- ▷ STEP 3: eliminate coefficients under pivot: $row_i \leftarrow row_i - \frac{a_{ik}}{a_{kk}} row_k$
- **for** $i = k+1 \ldots n-1$ **do**
  - Compute $m_{ik} = \frac{a_{ik}}{a_{kk}}$
  - $a_{ik} = 0$
  - **for** $j = k+1 \ldots n-1$ **do**
    - $a_{ij} \leftarrow a_{ij} - m_{ik} a_{kj}$
  - **for** $j = 0 \ldots n-1$ **do**
    - $b_{ij} \leftarrow b_{ij} - m_{ik} b_{kj}$

This can be followed by back-substitution.
Possible variation to limit the number of divisions:
Thanks to the row normalisation, the Algorithm 8 transforms A into an upper di-

Algorithm 8 Variation of Gaussian elimination with partial pivoting

for $k = 0 \dots n-1$ **do**
  Do STEP 1 and STEP 2 of Algorithm 7
  ▷ STEP 3: normalise row k: $row_k \leftarrow \frac{1}{a_{kk}} row_k$
  Compute $m_k = \frac{1}{a_{kk}}$
  $a_{kk} = 1$
  for $j = k+1 \dots n-1$ **do**
    $a_{kj} \leftarrow m_k a_{kj}$
  for $j = 0 \dots n-1$ **do**
    $b_{kj} \leftarrow m_k b_{kj}$
  ▷ STEP 4: eliminate coefficients under pivot $row_i \leftarrow row_i - a_{ik} row_k$
  for $i = k+1 \dots n-1$ **do**
    for $j = k+1 \dots n-1$ **do**
      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$
    for $j = 0 \dots n-1$ **do**
      $b_{ij} \leftarrow b_{ij} - a_{ik} b_{kj}$
    $a_{ik} = 0$

agonal matrix with the value 1 for every diagonal coefficient. The back-substitution that will follow will not include any divisions.

Gauss-Jordan algorithm with partial pivoting is very similar. The elimination is done not only on the coefficients under the pivot but also for the coefficients above the pivot. In STEP 3 of Algorithm 7 and in STEP 4 of Algorithm 8, the statement "**for** $i = k+1 \dots n-1$ **do**" is simply replaced by "**for** $i = \mathbf{0} \dots n-1$ **with** $i \neq k$ **do**". At the end of Gauss-Jordan elimination, A is changed into a diagonal matrix.

TRITA TRITA-ICT-EX-2017:6

www.kth.se