Automatic Generation of Hardware Designs for Matrix-Matrix Multiplication

ETH Zürich, Department of Computer Science

Master Thesis

Automatic Generation of Hardware Designs for Matrix-Matrix Multiplication

May 1, 2012

Author:
François Serre

Supervisors:
Pr. M.Püschel
Dr. M.Zuluaga

Abstract

Matrix-Matrix Multiplication (MMM) is a key computational kernel in scientific and engineering applications. Therefore, different implementations of this operation have been designed for FPGAs. However, it is hard to find, given a particular set of constraints (maximal number of slices, minimum frequency), the most appropriate design.

This thesis proposes the Operator Language for Schedules (OLS) to describe hardware designs that can perform MMM on arbitrarily-sized matrices. Algorithmic and implementation strategies such as blocking and reuse are represented as a set of rewriting rules that are recursively applied to OLS expressions. Finally, a compiler was built to translate a final OLS expression into Verilog code.

The different rewriting-rules that can be recursively applied, given the size requirements for MMM, give rise to large design space. Every design alternative, for a subset of matrix sizes, was synthesized and routed for our target FPGA platform and precise cost and performance metrics were stored in a database. Designers can later visualize the alternatives in our database and choose the design that suits the goals and constraints of the target system.

Résumé

La multiplication de matrices (MMM) est une opération omniprésente en sciences et en ingénierie. Il existe de ce fait de nombreuses implémentations pour FPGAs. Néanmoins, étant donné un ensemble de contraintes (nombre maximal de cellules logiques de chaque type à occuper, fréquence minimale), il est difficile de trouver la meilleure implémentation correspondante.

Cette thèse propose un langage pour décrire un design pour FPGA au travers d'une formule mathématique qui opère sur une matrice, qui elle-même représente un flux de données. Les différentes façons d'effectuer la multiplication par blocs, en réutilisant pour ce faire ou non le même module, sont décrites par un ensemble de règles de réécriture qui sont appliquées de manière récursive sur une formule de départ. La formule obtenue est ensuite traduite en Verilog.

L'espace des différentes implémentations de MMM ainsi généré est synthétisé et routé, puis la fréquence maximale et le nombre de cellules occupées de chaque sorte sont enregistrées dans une base de données. L'utilisateur n'a plus qu'à accéder à cette base au travers d'une interface web pour choisir l'implémentation qui convient le mieux à ses contraintes.

Acknowledgments

I would firstly like to thank my mentor, M. Pollefeys, for his guidance along my graduate studies at the ETH.

I would also like to thank M. Püschel for his ideas, his advice, and the motivation he had to understand all of my ideas.

I also owe a great amount of thanks to M. Zuluaga for her advice, her support (especially when I had some issues with the synthesis toolchain), her patience (I managed to crash her server six times in six months) and the time she spent re-reading this document.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

A lot of effort have been devoted to improving the performance of matrix-matrix multiplication (MMM) algorithms, as it is one of the most important operations in digital signal processing. Because of its structure and its high regularity, it is possible to parallelize MMM in many different ways. Therefore, for a given problem (size of the input matrices, platform), a lot of different implementations are possible, and due to the complexity of today's computers (Multi-core, SIMD instructions, cache hierarchy), finding the most appropriate one is not an obvious issue [2],[1].

This thesis focuses on hardware implementations of MMM to be used in customized platforms such as field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC). In these case, the problem is even more complex as the speed is not the only constraint. In fact, different MMM implementations will consume a different amount of hardware resources.

Another issue is that the matrix multiplier may have to be synchronous with some other processing elements. In this case, it is necessary to choose the implementation that has the best performance at a given frequency. In fact, the highest throughput may not be obtained with the same design depending on whether or not they can operate at their maximum frequency. And of course, if the implementation is not able to operate at the given frequency, it will slow down the whole system.

The hardware resources that a design requires, and its maximum frequency are determined after complex (synthesis) and non-deterministic (place and route) operations. Therefore, it is hard to find the best MMM implementation given those constraints.

1.2 Previous work

1.2.1 MMM Hardware Implementations

In [3] [11], a Matrix Multiplication algorithm is explained, but focuses mainly on an efficient implementation of an Floating Processing Unit. In our case, we consider 16 bits fixed-point operations. Another matrix implementation can be found in [5].

[12] describes an efficient way of implementing sparse matrix multiplication.

In its Linear Algebra Toolkit [10], Xilinx developped a tool able to generate a MMM for FPGAs. The user can set the dimentions of the input matrices, and a folding factor. However, this tool proposes only folding through rows, and generates a netlist that is only working with some Xilinx products.

1.2.2 Languages to describe an Algorithm

Spiral [7], [6] is a framework for the automatic generation of software and hardware libraries. It uses an internal language, SPL, and a rewriting rule system to describe linear algorithms from a high level. Operator Language [4] (OL) extends SPL to the non-linear domain. SPL and OL are high level languages which allow it to work for a large variety of platforms. Divide-and-conquer algorithms are described as breakdown rules that are recursively applied on a formula, the initial formula being the mathematical function to implement, or *kernel*. These formulas are composed of operators, n-ary functions that work on vectors. The most interesting element of these languages is the Kronecker product, which allows to identify easily the operations that are performed several times. When the formula obtained is fully expanded, a tagging operation takes place to assign the different parts of the formula to a specific platform element. However, an important post-processing is needed to obtain the final output, especially for hardware.

1.3 Thesis Overview

This thesis proposes a method to automatically generate a big variety of implementations for a given MMM problem. We only focus on the case where the whole input matrices are fed during one cycle, and the output matrix is returned during one cycle too. Those implementations are then synthesised and routed to fill a database with their caracteristics. Then, when a user wants an MMM design with given constraints, he can pick directly the best one from this database.

To describe the algorithms that our generator will produce, we will use the same formalism as the one described in [4]. However, to simplify the post-processing required on the final formulas, some new elements are added to the language. In the original OL, the operators work on vectors. The position of each element in these vectors represent an abstract position in the dataflow. In this thesis, a dataflow is represented by a matrix. The row of an element represents its position in an array, and its column represents the time (i.e. clock cycle) at which it is available. We also introduce two classes of operators:

- Combinational operators: outputs at a given cycle depend only on the current inputs. All of the operators introduced in [4] fall in this category.
- Sequential operators: outputs at a given cycle may depend on any previous input and/or the cycle number.

Using these two classes of operator, it is possible to describe all the circuit elements that are needed to perform MMM. Therefore, a terminated formula directly maps to circuitry, and the only post-processing needed is a translation of this formula into verilog.

To compute an MMM, we block it until we have only scalar multiplications and scalar additions. To do so, a small set of rewriting rules is used, to block the matrix multiplication in the three directions (vertically, horizontally and in depth), either parallely or serially. In the parallel case, all blocks are computed at the same time by a piece of circuitry implemented several times. In the serial case, all blocks are computed one after the other by the same piece of circuitry. The sequence of rules that we use defines a *design family*, and the set of parameters for these rules defines a *design*.

For a given problem (size of input matrices), the generator produces the verilog code that corresponds to a set of designs that are meant to cover a wide range of tradeoffs between cost and performance. This code is then synthesized in a distributed way, and the result of this synthesis (maximum frequency, area requirements) is collected in a central database. The user can finally choose the best design given his contraints on a web interface.

1.4 Notations

In this thesis, the tensor product (produces a $p+q$ order tensor) is noted $\odot$ to differenciate it from the Kronecker product $\otimes$ (produces a $max(p,q)$ order tensor). Otherwise, this document uses the notations used in [9]:

Einstein notation When an index variable appears twice or more times in a term, it means that it has to be summed over all its possible values:

$$c_i = a_{i,j,k}b_{j,k} \text{ means } c_i = \sum_j \sum_k a_{i,j,k}b_{j,k}$$

Order of a Tensor A tensor is underlined as many times as its order. A tensor which order is not specified is not underlined. Every indices will be at bottom.

1.5 Organization

This thesis is organized as follows. Chapter 2 introduces formally the formula language that our generator uses. Next, Chapter 3 describes how this language is used in the case of MMM and with Verilog. Then, Chapter 4 shows the results that our generator can get on a particular platform. Lastly, Chapter 5 presents concluding remarks.

2 Operator Language for Schedules

The Operator Language (OL), defined in [4], allows to describe the rules that, from a numerical kernel description, allow to derive an algorithm formula. However, this algorithm is still described at a high level, and heavy post-processing is needed to eliminate some remaining degrees of freedom (temporal reuse of some blocks, synchronous issues) before generating the final output. Plus, the algorithm is not aware of some timing issues: some data might not be available at all times.

The goal of this chapter is to extend the operator language (OL) so that the algorithms can easily deal with the temporal dimension. This extension is backward compatible with all the operators - called *combinational operators* - from OL, but introduces a new class of operators, the *sequential operators*, which can be used to describe sequential logic (flip/flops, multiplexers, accumulators...). Furthermore, the post-processing step that used to look for reused blocks to try to implement them sequentially is not needed anymore, since combinational operators naturally work every cycle.

However, this extension requires some knowledge on tensors that was not needed for OL (mainly the doubly contracted product). Therefore, a small reminder on tensors is included.

2.1 A reminder on tensors

2.1.1 Tensor

Definition

An n-order tensor T is a n-dimensional array[1]. We use the notation $t_{i_1,...,i_n}$ to represent its elements.

Examples

Table 2.1 lists the different kinds of tensor that are used in this document.

[1]This is a very cheap definition, but we don't need more

n	Notation	Usual name	Elements	Equivalent in C
0	$s \in \mathbb{C}$	Scalar	s	double s;
1	$\underline{v} \in \mathbb{C}^k$	Vector	$(v_0, ..., v_{k-1})$	double v[k];
2	$\underline{\underline{M}} \in \mathcal{M}_{k,l}(\mathbb{C})$	Matrix	$(m_{i,j})_{0 \leq i \leq k-1, 0 \leq j \leq l-1}$	double M[k][l];
4	$\underline{\underline{\underline{\underline{C}}}}$		$(c_{i,j,k,l})_{i,j,k,l}$	double C[a][b][c][d];

Table 2.1: Examples of order n tensors

2.1.2 Canonical basis

The notation $\underline{e_{i_1}} \odot \underline{e_{i_2}} \odot ... \odot \underline{e_{i_n}}$ represents the n-order tensor that contains a 1 at position $(i_1, i_2, ..., i_n)$, and zeroes elsewhere. It is easy to check that $(\underline{e_{i_1}} \odot ... \odot \underline{e_{i_n}})_{i_1,...,i_n}$ is an orthonormal basis of the set of order n tensors[2]. Therefore, every tensor T can be rewritten as:

$$T = t_{i_1,...,i_n}\underline{e_{i_1}} \odot ... \odot \underline{e_{i_n}}$$

Examples

- The vector $\begin{pmatrix}1\\0\\3\end{pmatrix}$ can be rewritten as $\underline{e_0} + 3\underline{e_2}$.
- The matrix $\begin{bmatrix}1 & 0\\2 & 5\end{bmatrix}$ can be rewritten as $\underline{e_0} \odot \underline{e_0} + 2\underline{e_1} \odot \underline{e_0} + 5\underline{e_1} \odot \underline{e_1}$

2.1.3 Tensor Product

Definition

If A is a p-order tensor, and B a q-order tensor, the tensor product $A \odot B$ is the $p+q$ order tensor such that:

$$A \odot B = (a_{i_1,...,i_p} \times b_{i_{p+1},...,i_{p+q}})\underline{e_{i_1}} \odot ... \odot \underline{e_{i_{p+q}}}$$

Examples

- The tensor product of two vectors $\underline{u} = u_i\underline{e_i}$ and $\underline{v} = v_j\underline{e_j}$ is the matrix:

$$\underline{u} \odot \underline{v} = (u_i \times v_j)\underline{e_i} \odot \underline{e_j} = \underline{u} \cdot {}^t\underline{v}$$

- The tensor product of two matrices $\underline{\underline{M}} = m_{i,j}\underline{e_i} \odot \underline{e_j}$ and $\underline{\underline{N}} = n_{k,l}\underline{e_k} \odot \underline{e_l}$ is the order 4 tensor[3]:

$$\underline{\underline{M}} \odot \underline{\underline{N}} = (m_{i,j} \times n_{k,l})\underline{e_i} \odot \underline{e_j} \odot \underline{e_k} \odot \underline{e_l}$$

- It is easy to check that the tensor product of an element of the canonical basis of the p-order tensors and one of the canonical basis of the q-order tensors gives the corresponding element of the canonical basis of the $p+q$-order tensors. This justifies à postérioris our notation $\underline{e_{i_1}} \odot ... \odot \underline{e_{i_n}}$ for an element of the canonical basis.

[2]If we consider it as an hermitian vector space with the element-wise addition, the element-wise multiplication by a scalar, and the sum of the element-wise conjugated products as a scalar product.

[3]This makes the tensor product defined here different from the Kronecker product used in OL. The last takes two matrices, and outputs a bigger matrix. In this document, the tensor product is used to clearly separate the temporal dimension from the "position" dimension.

2.1.4 Simply Contracted Product

Definition

If A is a p-order tensor, and B a q-order tensor, the simply contracted product (aka contracted product) $A.B$ is the $p+q-2$-order tensor such that:

$$A.B = (a_{i_1,\ldots,i_p-1,k} \times b_{k,i_{p+2},\ldots,i_{p+q}})\underline{e_{i_1}} \odot \ldots \odot \underline{e_{i_{p-1}}} \odot \underline{e_{i_{p+2}}} \odot \ldots \odot \underline{e_{i_{p+q}}}$$

In other words, the contracted product of two tensors works likewise the tensor product, except that the last dimension of the first tensor is summed along the first dimension of the second tensor[4].

Examples

- The contracted product of two vectors $\underline{u} = u_i\underline{e_i}$ and $\underline{v} = v_j\underline{e_j}$ is the scalar:

  $$\underline{u} \cdot \underline{v} = u_i v_i$$

  In other words, the contracted product of two vectors is the canonical scalar product of $\mathbb{R}^k$.

- The contracted product of a matrix $\underline{\underline{M}} = m_{i,j}\underline{e_i} \odot \underline{e_j}$ and a vector $\underline{u} = u_k\underline{e_k}$ is the vector:

  $$\underline{\underline{M}} \cdot \underline{u} = m_{i,j}u_j\underline{e_i}$$

  This is the classical Matrix-Vector multiplication.

- The contracted product of two matrices $\underline{\underline{M}} = m_{i,j}\underline{e_i} \odot \underline{e_j}$ and $\underline{\underline{N}} = n_{k,l}\underline{e_k} \odot \underline{e_l}$ is the matrix:

  $$\underline{\underline{M}} \cdot \underline{\underline{N}} = (m_{i,j}n_{j,l})\underline{e_i} \odot \underline{e_l}$$

  This is the classical Matrix-Matrix multiplication.

2.1.5 Doubly Contracted Product

Definition

If A is a p order tensor, and B a q order tensor, the doubly contracted product $A : B$ is the $p+q-4$ order tensor such that:

$$A : B = (a_{i_1,\ldots,i_p-2,k,l} \times b_{l,k,i_{p+3},\ldots,i_{p+q}})\underline{e_{i_1}} \odot \ldots \odot \underline{e_{i_{p-2}}} \odot \underline{e_{i_{p+3}}} \odot \ldots \odot \underline{e_{i_{p+q}}}$$

In other words, the doubly contracted product of two tensors works similarly to the simply contracted product, except that the last but one dimension of the first tensor is summed along the second dimension of the second tensor[5].

[4] The size of the last dimension of the first tensor must equal the size of the first dimension of the second tensor!

[5] The size of the last dimension of the first tensor must equal the size of the first dimension of the second tensor, AND the size of the one but last dimension of the first tensor must equal the size of the second dimension of the second tensor!

Examples

- The doubly contracted product of two matrices $\underline{\underline{M}} = m_{i,j}\underline{e_i} \odot \underline{e_j}$ and $\underline{\underline{N}} = n_{k,l}\underline{e_k} \odot \underline{e_l}$ is the scalar:
$$\underline{\underline{M}} : \underline{\underline{N}} = m_{i,j}n_{j,i} = Tr(\underline{\underline{M}} \cdot \underline{\underline{N}})$$
If $\underline{\underline{M}}$ (and $\underline{\underline{N}}$) is square, then ${}^t\underline{\underline{M}} : \underline{\underline{N}}$ defines the classical scalar product on $\mathcal{M}_k(\mathbb{R})$

- If $\underline{\underline{M}}$ is square, and if $\underline{\underline{I}}$ is the identity matrix ($\underline{\underline{I}} = \underline{e_i} \odot \underline{e_i}$), we have:
$$\underline{\underline{M}} : \underline{\underline{I}} = m_{i,j}\delta^i_j = m_{i,i} = Tr(\underline{\underline{M}})$$

- The doubly contracted product of an order 4 tensor $\underline{\underline{\underline{\underline{C}}}} = c_{i,j,k,l}\underline{e_i} \odot \underline{e_j} \odot \underline{e_k} \odot \underline{e_l}$ and a matrix $\underline{\underline{M}} = m_{i,j}\underline{e_i} \odot \underline{e_j}$ is a matrix :
$$\underline{\underline{\underline{\underline{C}}}} : \underline{\underline{M}} = c_{i,j,k,l}m_{l,k}\underline{e_i} \odot \underline{e_j}$$

2.2 Schedules

The main difference between the original Operator Language and the language defined here is the type of data the operators work on. The operators in OL work on vectors such as $v_i\underline{e_i}$. v_i is the value contained at the index i. This index is an abstraction in the sense that its definition is left to the user (memory address, value of a register at a given time,...).

In SOL, operators work on schedules. A schedule is a matrix
$$\underline{\underline{s}} = s_{i,t}\underline{e_i} \odot \underline{e_t}$$
$s_{i,t}$ is the value contained at the position i (this position can be a port, a bus, or a memory address) at time t (typically, a clock cycle for synchronous hardware).

2.3 Operators

The operators are the elements that work on schedules (cf. Fig. 2.1).

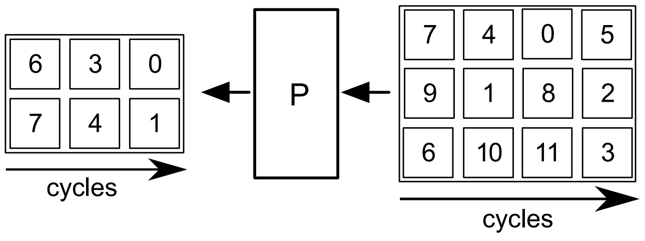

Figure 2.1: An operator working on a dataflow

2.3.1 Combinational Operators

A combinational operator is an operator for which the output at cycle t is a pure function of its input at cycle t. This typically describes a combinational circuit. All operators described in [4] enter in this category.

Linear combinational operators with arity $(1,1)$

A linear combinational operator with arity $(1,1)$ can be represented by a matrix $\underline{\underline{M}}$(the same as in the OL paper). The recipe to apply it on a schedule $\underline{\underline{s}}$ is the same as for a vector; a simply contracted product:

$$\underline{\underline{r}} = \underline{\underline{M}} \cdot \underline{\underline{s}}$$

Examples

- The $N \times N$ matrix
$$\underline{\underline{DFT_N}} = e^{-\frac{2i\pi}{N}(j-1)(k-1)} \underline{e_j} \odot \underline{e_k}$$
is the discrete Fourier transform. Applying it on a $N \times T$ matrix will perform T Fourier transforms (1 per cycle).

- We note:
$$\underline{\underline{ADD}} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$
This typically describes an adder (adds two numbers every cycle, cf. Fig. 2.2).

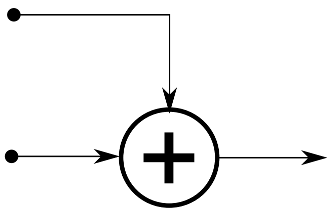

Figure 2.2: Implementation of $\underline{\underline{ADD}}$

Other Combinational Operators

This definition naturally extends to non-linear and/or with non-trivial arity operator. For instance, let B: $\mathbb{C}^I \times \mathbb{C}^{I'} \to \mathbb{C}^K$ be any operator. We define its action on schedules:

$$\underline{\underline{B\left(\underline{\underline{s_1}}, \underline{\underline{s_2}}\right)}} = \underline{B\left(\underline{\underline{s_1}}.\underline{e_t}, \underline{\underline{s_2}}.\underline{e_t}\right)} \odot \underline{e_t}$$

Examples

- We note $MMM_{m,k,n}$ the matrix multiplication that consumes two matrices and produces one:
$$MMM_{m,k,n} : \mathbb{C}^{mk} \times \mathbb{C}^{kn} \to \mathbb{C}^{mn}; (A, B) \mapsto AB$$
Applying it on two schedules $mk \times T$ and $kn \times T$ will perform T matrix multiplications (1 per cycle). The output will be a $mn \times T$ schedule.

- We define
$$MUL = MMM_{1,1,1}$$
This typically describes a multiplier (multiply two numbers every cycle).

2.3.2 Sequential Operators

An operator that is not combinational is sequential.

Linear with arity $(1, 1)$

Sequential operators can access the time dimension. Therefore, linear sequential operators with arity $(1, 1)$ use doubly contracted product, and are order 4 tensors. Let's doubly contract an order 4 tensor $\underline{\underline{\underline{\underline{T}}}} = t_{i',t',t,i}\underline{e_{i'}} \odot \underline{e_{t'}} \odot \underline{e_t} \odot \underline{e_i}$ with a schedule $\underline{\underline{s}} = s_{i,t}\underline{e_i} \odot \underline{e_t}$:

$$\underline{\underline{\underline{\underline{T}}}} : \underline{\underline{s}} = t_{i',t',t,i}s_{i,t}\underline{e_{i'}} \odot \underline{e_{t'}}$$

The meaning of $t_{i',t',t,i}$ is clear: it is the contribution of the value of the input at time t and at position i on the output at time t' and position i'.

Examples

- We note $\underline{\underline{\underline{\underline{D_c}}}}$ the operator:
$$\underline{\underline{\underline{\underline{D_c}}}} = \underline{e_i} \odot \underline{e_{t+c}} \odot \underline{e_t} \odot \underline{e_i}$$
Let's apply $\underline{\underline{\underline{\underline{D_c}}}}$ on a schedule:
$$\underline{\underline{\underline{\underline{D_c}}}} : \underline{\underline{s}} = s_{i,t}\underline{e_i} \odot \underline{e_{t+c}}$$
The whole schedule has been delayed of c cycles. This typically describes a buffer of size c.

- Let k be a positive integer, and u a sequence in $[\![0; k-1]\!]^{\mathbb{N}}$ Then, we denote:
$$\underline{\underline{\underline{\underline{MUX_u}}}} = \underline{e_0} \odot \underline{e_t} \odot \underline{e_t} \odot \underline{e_{u(t)}}$$
When this operator is applied to a schedule, it returns at cycle t the value contained at position u_t. This describes the behavior of a multiplexer controlled by a cycle counter. As we only consider data-independent kernels, this is actually the only kind of multiplexer that we can find(cf. Fig. 2.3).

- Let h be a positive integer. We denote:
$$\underline{\underline{\underline{\underline{T_h}}}} = \underline{e_i} \odot \underline{e_{h\times t}} \odot \underline{e_{h\times t}} \odot \underline{e_i}$$
At cycle t, this operator returns its input if t is a multiple of h, and zeroes otherwise.

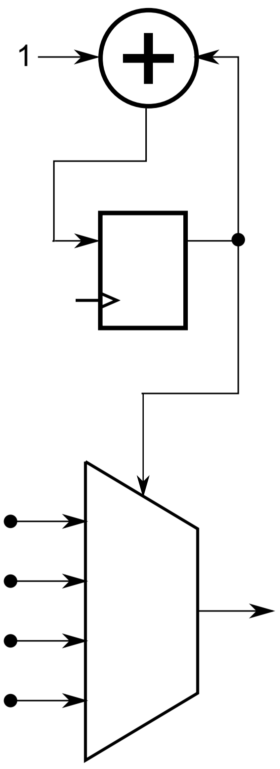

Figure 2.3: Implementation of $\underline{\underline{\underline{MUX_u}}}$

Non Linear

This is the most general operator. It takes one or more schedules, and returns one or more schedules.

Example $MMM^{T,\delta}_{m,k,n}$ is the operator that multiplies a $m \times k$ and a $k \times n$ matrix with a gap of T cycles, and a latency of $T + \delta$ cycles.

2.3.3 Composition

We denote $A \circ B$ the composition of operators A and B (cf Fig. 2.4).

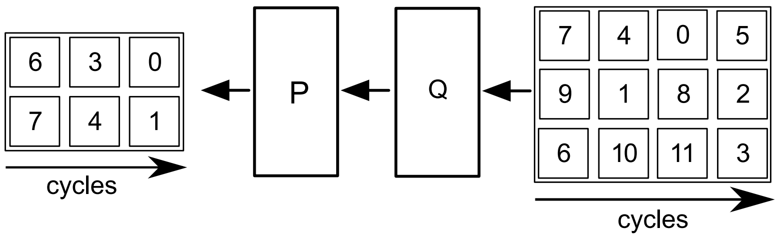

Figure 2.4: Composition of two Operators

Linear operator with arity $(1, 1)$ can easily be composed using the two kinds of contracted products.

Two Linear Combinational Operators

As the simply contracted product is associative, we have:

$$\underline{\underline{A}} \cdot (\underline{\underline{B}} \cdot \underline{\underline{s}}) = (\underline{\underline{A}} \cdot \underline{\underline{B}}) \cdot \underline{\underline{s}}$$

This means that the composition of two linear combinational operators with arity $(1,1)$ is the matrix product of those two operators.

Two Linear Sequential Operators

It is easy to show that the doubly contracted product is associative. Therefore:

$$\underline{\underline{\underline{\underline{A}}}} : (\underline{\underline{\underline{\underline{B}}}} : \underline{\underline{s}}) = (\underline{\underline{\underline{\underline{A}}}} : \underline{\underline{\underline{\underline{B}}}}) : \underline{\underline{s}}$$

This means that the composition of two linear sequential operators with arity $(1,1)$ is the doubly contracted product of those two operators.

A mix between linear non-combinational and linear combinational operators

The following equalities are easy to check:

$$\underline{\underline{A}} \cdot (\underline{\underline{\underline{\underline{B}}}} : \underline{\underline{s}}) = (\underline{\underline{A}} \cdot \underline{\underline{\underline{\underline{B}}}}) : \underline{\underline{s}}$$

$$\underline{\underline{\underline{\underline{A}}}} : (\underline{\underline{B}} \cdot \underline{\underline{s}}) = (\underline{\underline{\underline{\underline{A}}}} \cdot \underline{\underline{B}}) : \underline{\underline{s}}$$

This means that the composition of a mix of linear non-combinational and combinational operators with arity $(1,1)$ is the simply contracted product of those two operators. The resulting operator is a non-combinational one.

Examples

- We denote

  $$\underline{\underline{\underline{\underline{P_{l,h}}}}} = \underline{\underline{\underline{\underline{D_l}}}} : \underline{\underline{\underline{\underline{T_h}}}}$$

  Now, let O be any combinational operator. Then, $\underline{\underline{\underline{\underline{P_{l,h}}}}} \circ O$ is the operator that performs O every h cycles with a latency of l cycles. For instance:

  $$\underline{\underline{\underline{\underline{P_{4,2}}}}} \cdot \underline{\underline{DFT_N}}$$

  represents the pipelined version of the DFT that has a throughput of one DFT per 2 cycles, and a latency of 4.

2.3.4 Kronecker Product

The Kronecker product $\otimes$ is the most important higher order operator. It is defined as follows for arity $(1,1)$ linear combinational operators:

$$\underline{\underline{A}} \otimes \underline{\underline{B}} = [a_{i,j}B], \underline{\underline{A}} = a_{i,j}\underline{e_i} \odot \underline{e_j}$$

In the case where $\underline{\underline{A}} = \underline{\underline{I_n}}$, we have :

$$\underline{\underline{I_n}} \otimes \underline{\underline{B}} = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix}$$

In this case, B is applied to a concatenation of n input schedules.

OL extends this to more general combinational operators. We define the same way the Kronecker product of a multi-linear operator $A : \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}^r$ and any sequential operator $B : \mathcal{M}_{m,T}(\mathbb{R}) \times \mathcal{M}_{n,T}(\mathbb{R}) \to \mathcal{M}_{k,T'}(\mathbb{R})$:

$$(A \otimes B)(\underline{\underline{x}}, \underline{\underline{y}}) = A(\underline{e_i}, \underline{e_j}) \otimes B\left((\underline{e_0} \odot \underline{e_i}) \otimes \underline{\underline{I_m}}) \cdot \underline{\underline{x}}), (\underline{e_0} \odot \underline{e_j}) \otimes \underline{\underline{I_n}}) \cdot \underline{\underline{y}})\right)$$

This definition holds for any combinational operator using the way we defined their action on schedules.

Intuitively, a Kronecker product works as follows: the input schedules are blocked horizontally into p (resp. q) schedules of m (resp. n) rows. Then, A controls the way these sub-schedules are fed to several instances of B, and how the chunks of schedules produced have to be combined. Note that nothing changed in the temporal dimension.

2.4 Rules

Now that we have defined the elements that compose a formula, we will define the way we manipulate them.

A *rewriting rule* $\mathcal{R}$ is an application that takes a formula, searches for a given pattern in this formula, and replaces it with an equivalent expression. In the case where the given pattern appears several times, only the first one (at the lowest level of the formula) encountered is replaced. For instance, the rule:

$$MMM^{0,1}_{1,1,1} \to \underline{\underline{\underline{D_1}}} \circ MUL$$

replaces the first occurrence of $MMM^{0,1}_{1,1,1}$ in a formula by the expanded version $\underline{\underline{\underline{D_1}}} \circ MUL$.

A pattern can contain variables. In this case, those variables are replaced in the replacement formula:

$$\underline{\underline{I_i}} \circ Expr \to Expr$$

The above rule cleans a formula by removing one eventual useless identity operator. Table 2.2 shows some similar rules used to clean up formulas.

Rules can be composed. We denote $\mathcal{CR}$ the set of cleaning rules applied in sequence.

If, for any formula, the same rule $\mathcal{R}$ applied several times returns the same formula after a certain time, we denote $\mathcal{R}^\infty$ the corresponding application. This is the case if the replacement formula does not contain the pattern. For instance,

$$(\underline{\underline{I_1}} \otimes Expr \to Expr)^\infty$$

will remove all Kronecker products where the left term is $\underline{\underline{I_1}}$ from a formula.

To generate an implementation, the generator takes a kernel description, applies a set of breakdown rules on it until it has a terminated formula. The set of breakdown rules that we use for MMM is described in the next chapter. After each application of a breakdown rule, $\mathcal{CR}^\infty$ is applied on the resulting formula.

First cleaning rule set
Removal of useless kronecker products :
$\underline{\underline{I_1}} \otimes Expr \rightarrow Expr$
$\underline{\underline{I_i}} \otimes \underline{\underline{I_j}} \rightarrow \underline{\underline{I_{ij}}}$
Removal of useless identity operators :
$\underline{\underline{I_i}} \circ Expr \rightarrow Expr$
$Expr \circ \underline{\underline{I_i}} \rightarrow Expr$
$Expr \circ \left(\underline{\underline{I_i}} \times \underline{\underline{I_j}}\right) \rightarrow Expr$
$\left(\underline{\underline{I_i}} \times \underline{\underline{I_j}}\right) \circ Expr \rightarrow Expr$
Factorisation of cross product :
$(A \times B) \circ (C \times D) \rightarrow (A \circ C) \times (B \circ D)$

Table 2.2: Cleaning rules

3 Matrix-Matrix Multiplication in Verilog

3.1 Formula to Verilog

Before going further, we have to define what is a terminated formula for Verilog. In other words, we have to list the operators that can be translated to verilog, and to describe how the operations on operators work.

3.1.1 Composition

When a terminated formula is translated into a Verilog module, composed operators are simply translated one by one; the output of an operator constitutes the input of the following operator. The right-most operator receives the module's inputs, and the outputs of the left-most operators are the module's output.

For instance, the formula $A \circ B \circ C$ would be translated into:

```
module mod(input clk, input i1, input i2, output o1, output o2);
        wire tmp1; //output for C
        C(clk, i1, i2, tmp1); //Operator C

        wire tmp2, tmp3; //outputs for B
        B(clk, tmp1, tmp2, tmp3); //Operator B

        wire tmp4, tmp5; //outputs for C
        A(clk, tmp2, tmp3, tmp4, tmp5); //Operator C

        assign o1=tmp4; //Assign the output of C to the module
        assign o2=tmp5;
endmodule
```

3.1.2 Combinational Operators

Permutations

Permutations are the simplest operators since they don't involve any verilog code. They simply change the order of the inputs of the following operator.

Transposition The transposition operator, noted $\underline{\underline{L_m^{mn}}}$ is the arity (1,1) permutation that transposes a $m \times n$ matrix (seen as a linearized vector). Therefore:

$$\underline{\underline{L_m^{mn}}} \cdot \underline{e_{i\times n+j}} = \underline{e_{j\times n+i}}$$

Swap The swap operator, noted $SWAP$ is the arity (2,2) operator that simply exchanges its inputs:

$$SWAP\left(\underline{x}, \underline{y}\right) = \left(\underline{y}, \underline{x}\right)$$

Multiplier

A multiplier is an arity (2,1) operator that simply multiplies two scalars. It is noted $MMM_{1,1,1}$:

$$MMM_{1,1,1}(a, b) = ab$$

Its implementation is straightforward:

```
assign out = in1 * in2;
```

3.1.3 Sequential Operators

D-type Flip-flop

The D-type flip-flop is the simplest sequential operator. It is represented by the formula:

$$\underline{\underline{\underline{\underline{D_1}}}} = \underline{e_i} \odot \underline{e_{t+1}} \odot \underline{e_t} \odot \underline{e_i}$$

At each cycle, the value returned is the value present at the input during the previous cycle (cf. Fig. 3.1).

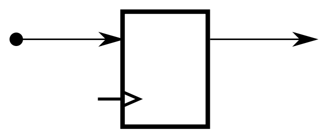

Figure 3.1: Implementation of $\underline{\underline{\underline{\underline{D_1}}}}$

The corresponding Verilog code is the following:

```
reg [15:0] out;
always @(posedge clk)
    out <= in;
```

Counters and Triggers

The following operators require to receive a signal at a regular time. Therefore, each of them requires a counter that counts cycles for a given period, and a trigger that fires when this counter reaches a particular value. However, having one counter for each operator would wastefully occupy ressources. Similarly, some triggers might fire exactly at the same time for several operators. Consequently, one global set of counter and triggers is created when at least one operator requires it.

For instance, let's assume that an operator requires a trigger that fires each cycle congruent to 0 modulo 4, and another one that needs a signal each cycle congruent to 3 modulo 6. The translator will generate a counter that counts until $gcd(4, 6) = 12$, one trigger that fires when the counter reaches the values $\{0, 4, 8\}$ (in this case, the synthetiser is intelligent enough to only watch the first two bits), and another one for values $\{3, 9\}$.

When a counter is present in a module, an input signal *start* appears. This signal has to be set by the user when he begins to feed the data. This resets the counter and synchronises all the operators.

```
reg [3:0] counter;
always @(posedge clk)
        if (start || counter == 12)
                counter <= 0;
        else
                counter <= counter + 1;
assign t0 =
        (counter == 0) ||
        (counter == 4) ||
        (counter == 8);
assign t1 =
        (counter == 3) ||
        (counter == 9);
```

As the presence of a trigger implies non-trivial additions in the formulas, the next mathematical definitions in this section will not use Einstein's notation.

Memory

The memory operator has one input and one output. It stores its input at cycle φ modulo T, and holds this value at the output until it gets a new one.

$$\underline{\underline{\underline{MEM_{\varphi,T}}}} = \sum_{\substack{0\leq i<T \\ t\equiv\varphi[T]}} \underline{e_0} \odot \underline{e_{t+i}} \odot \underline{e_t} \odot \underline{e_0}$$

Using a trigger t that fires at cycle φ modulo T, the Verilog implementation is straightforward:

```
reg [15:0] out;
wire [15:0] nextvalue;
assign nextvalue= t ? in : out;
always @(posedge clk)
        out <= nextvalue;
```

Accumulator

The accumulator is a linear operator with a scalar input and a scalar output. At every cycle the output value is the sum of the input value and the previous input values since its last reset. This operator can also work intermittently. A parameter δ specifies the number of cycles to wait between every additions. The other cycles are simply skipped:

$$\underline{\underline{\underline{ACC_{\varphi,T,\delta}}}} = \sum_{\substack{0\leq i<T \\ t\equiv\varphi[T\delta]}} \underline{e_1} \odot \underline{e_{t+T\delta}} \odot \underline{e_{t+i\delta}} \odot \underline{e_1}$$

The Verilog implementation uses two triggers: $t0$, that fires every cycle φ modulo δ, and $t1$ every cycle φ modulo $T\delta$.

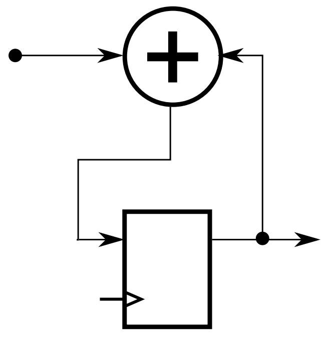

Figure 3.2: Implementation of $\underline{\underline{\underline{\underline{ACC}}}}$

```
reg [15:0] out;
wire [15:0] var;
assign var = t0 ? in : out + in;
always @(posedge clk)
        out <= t1 ? var : out;
```

Serializer

The serializer operator is an arity (1,1) linear operator. It takes a dimention n vector as an input, and has a scalar output. It stores its input at cycle φ modulo $n \cdot \delta$, and outputs its components one after the other during δ cycles:

$$\underline{\underline{\underline{\underline{SER_{n,\varphi,\delta}}}}} = \sum_{\substack{0 \le i < n \\ 0 \le j < \delta \\ t \equiv \varphi [n\delta]}} \underline{e_1} \odot \underline{e_{t+i\delta+j}} \odot \underline{e_t} \odot \underline{e_i}$$

The serializer is implemented using a shift register. Once again, two triggers are used in the Verilog implementation: $t1$ to wait until the next δ cycle, and $t2$ to propagate the values through the next bit. Here is how a 3-inputs serializer would be written:

```
reg [15:0] var1;
always @(posedge clk)
        var1 <= t1 ? in0 : var1;
wire [15:0] var2;
assign var2 = t0 ? in1 : var1;
reg [15:0] var3;
always @(posedge clk)
        var3 <= t1 ? var2 : var3;
wire [15:0] var4;
assign out = t0 ? in2 : var3;
```

Deserializer

The deserializer is the opposite of the serializer. It takes a scalar input at every cycle as part of an input stream. The output is a n-dimentional vector that contains these values.

Operator	Name	Definition
$\underline{\underline{I_n}}$	Identity	$\mathbb{R}^n \to \mathbb{R}^n; \underline{x} \mapsto \underline{x}$
H_n	Hadamar product	$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n; (\underline{x}, \underline{y}) \mapsto x_i y_i \underline{e_i}$
S_n	Scalar product	$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}; (\underline{x}, \underline{y}) \mapsto x_i y_i$
$K_{m,n}$	Kronecker product	$\mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{mn}; (\underline{x}, \underline{y}) \mapsto x_i y_j \underline{e_{ij}}$

Table 3.1: Possible left term for a Kronecker product in a terminated formula

As the previous operators, a parameter δ allows to freeze the process during a few cycles:

$$\underline{\underline{\underline{\underline{DESER_{n,\varphi,\delta}}}}} = \sum_{\substack{0 \le i < n \\ 0 \le j < \delta \\ t \equiv \varphi[\delta]}} \underline{e_{n-i}} \odot \underline{e_{t+i\delta+j}} \odot \underline{e_t} \odot \underline{e_0}$$

The verilog implementation uses a shift register, and only one trigger t[1].

```
reg [15:0] var1;
always @(posedge clk)
        var1 <= t ? in : var1;
reg [15:0] var2;
always @(posedge clk)
        var2 <= t ? var1 : var2;
assign out0 = var2;
assign out1 = var1;
assign out2 = in;
```

3.1.4 Kronecker Products

The Kronecker product is the most complicated operation to implement. Therefore, only a very limited subset of cases can be translated to verilog. The left term of the Kronecker product must be in the list given in table 3.1 for a formula to be terminated.

When the translator encounters a Kronecker product, $Op \otimes Expr$, it first implements *Expr* into a new module. Then, this new module is called from the current module depending on *Op*:

Identity

If $Op = \underline{\underline{I_n}}$, the submodule *Expr* is called n times from the current module, each time consumming a subset of the inputs and of the outputs.

The example below shows how $\underline{\underline{I_3}} \otimes Expr$ would be translated.

```
wire [15:0] out0,out1,out2;
Expr(.clk(clk),.i0(in0),.i1(in1),.o(out0));
Expr(.clk(clk),.i0(in2),.i1(in3),.o(out1));
Expr(.clk(clk),.i0(in4),.i1(in5),.o(out2));
```

[1]This implementation does not conform to the mathematical definition. The output in a cycle that is not congruent with φ modulo $n\delta$ is not null. To get a correct implementation, an other trigger and a AND gate are necessary.

Hadamar Product Operator

The Kronecker product with the Hadamar product operator works similarly to the identity, but with arity $(2, 1)$ expressions. $H_n \otimes Expr$ calls $Expr$ n times, with components from the first input vector, and the corresponding components from the second input vector.

The example below shows how $H_3 \otimes Expr$ would be translated.

```
wire [15:0] out0,out1,out2;
Expr(.clk(clk),.x(x0),.y(y0),.o(out0));
Expr(.clk(clk),.x(x1),.y(y1),.o(out1));
Expr(.clk(clk),.x(x2),.y(y2),.o(out2));
```

Scalar Product Operator

With the scalar product, the translator will simply implement the Kronecker product with the Hadamar product, but will sum up the outputs at the end, using:

```
wire [15:0] out;
assign out = out0 + out1 +out2;
```

Kronecker Product Operator

Lastly, the Kronecker product with the Kronecker product operator. This time, all possible combinations of the input are used.

A $K_{2,2} \otimes Expr$ would be implemented:

```
wire [15:0] out0,out1,out2;
Expr(.clk(clk),.x(x0),.y(y0),.o(out0));
Expr(.clk(clk),.x(x0),.y(y1),.o(out1));
Expr(.clk(clk),.x(x1),.y(y0),.o(out2));
Expr(.clk(clk),.x(x1),.y(y1),.o(out3));
```

3.1.5 New Cleaning Rules

The following cleaning rules support the operators described in this section:

3.2 Specific Rules for MMM

The MMM algorithm can be blocked in three different ways (cf.Fig. 3.3).

3.2.1 Vertical and Horizontal Blocking

The vertical and horizontal blocking are the easiest to implement. One of the input matrices is split, and each part is multiplied with the second matrix. The final matrix is obtained by concatenating the results.

Rule	Signification
$\underline{\underline{L_1^i}} \to \underline{\underline{I_i}}$	Remove useless transpositions
$\underline{\underline{L_i^i}} \to \underline{\underline{I_i}}$	
$\underline{\underline{L_i^{ij}}} \cdot \underline{\underline{L_j^{ij}}} \to \underline{\underline{I_{ij}}}$	Transposition is involutive
$\underline{\underline{\underline{\underline{SER_{1,i,j}}}}} \to \underline{\underline{I_1}}$	Remove useless serializer
$\underline{\underline{\underline{\underline{DESER_{1,i,j}}}}} \to \underline{\underline{I_1}}$	Remove useless deserializer
$\underline{\underline{\underline{\underline{ACC_{i,1,j}}}}} \to \underline{\underline{MEM_{i,j}}}$	A one time accumulator is a memory
$\underline{\underline{\underline{\underline{MEM_{i,1}}}}} \to \underline{\underline{I_1}}$	Remove useless memories
$SWAP \circ SWAP \to \underline{\underline{I}}$	Swap is involutive
$S_1 \otimes Expr \to Expr$ $K_{1,1} \otimes Expr \to Expr$ $H_1 \otimes Expr \to Expr$	Remove useless kronecker products

Table 3.2: Additional cleaning rules

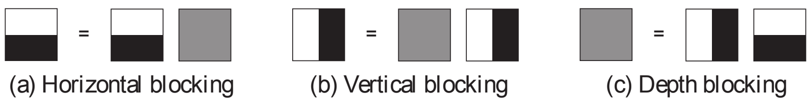

Figure 3.3: (From [4]) Blocking matrix multiplication along each one of the three dimensions. For the horizontal and vertical blocking, the white (black) part of the result is computed by multiplying the white (black) part of the blocked input with the other, gray, input. For the depth blocking, the result is computed by multiplying both white parts and both black parts and adding the results.

Parallel Horizontal Blocking

The horizontal blocking shows the power of the Kronecker product in the case of a parallel implementation. This rules takes a parameter m', the number of rows that the sub-block has:

$$\mathcal{R}_{m'}^{H,//} : MMM_{m,k,n}^{T,\delta} \to K_{m/m',1} \otimes MMM_{m',k,n}^{T,\delta} \tag{3.1}$$

Serial Vertical Blocking

The serial vertical blocking is done with the following rule:

$$\mathcal{R}_{n'}^{V,\cdots} : MMM_{m,k,n}^{t,\delta} \rightarrow \begin{cases} \left(\underline{\underline{I_{mn'}}} \otimes \underline{\underline{\underline{\underline{DESER_{n/n',tn'/n+\delta-1,tn'/n}}}}}\right) \circ \\ MMM_{m,k,n'}^{tn'/n,\delta} \circ \\ \left(\left(\underline{\underline{I_{mk}}} \otimes \underline{\underline{\underline{\underline{MEM_{0,t}}}}}\right) \times \left(\underline{\underline{I_{kn'}}} \otimes \underline{\underline{\underline{\underline{SER_{n/n',0,tn'/n}}}}}\right)\right) \end{cases} \tag{3.2}$$

Columns of the right input matrix are serialized while the left one is hold into registers. After the multiplications, the columns of the final matrix are deserialized.

Transposed Product

So far, we have a serial vertical blocking rule $\mathcal{R}_{n'}^{V,\cdots}$, and a parallel horizontal blocking rule $\mathcal{R}_{m'}^{H,//}$. A parallel vertical blocking rule and a serial horizontal blocking rule exist, but we will use instead the fact that:

$$\underline{\underline{A}} \cdot \underline{\underline{B}} = {}^{t}\left({}^{t}\underline{\underline{B}} \cdot {}^{t}\underline{\underline{A}}\right)$$

This leads to the rule:

$$\mathcal{R}^{T} : MMM_{m,k,n}^{T,\delta} \rightarrow \underline{\underline{L_{n}^{mn}}} \circ MMM_{n,k,m}^{T,\delta} \circ SWAP \circ \left(\underline{\underline{L_{m}^{mk}}} \times \underline{\underline{L_{k}^{kn}}}\right)$$

As this rule only contains permutations, no additional verilog code will be generated. Using this rule, and the two previous ones, we obtain all possible horizontal and vertical blocking rules.

3.2.2 Depth Blocking

The depth blocking is the last kind of blocking. The input matrices are split into columns for the left one, and into rows for the right one. Then, the columns are multiplied with the rows. Resulting matrices are added element-wize to form the final matrix.

Parallel

The whole parallel depth blocking can be captured by a kronecker product with a scalar product operator:

$$\mathcal{R}_{k'}^{D,//} : MMM_{m,k,n}^{T,\delta} \rightarrow \left(S_{k/k'} \otimes MMM_{m,k',n}^{T,\delta}\right) \circ \left((\underline{\underline{L_{k/k'}^{mk/k'}}} \otimes \underline{\underline{I_{k'}}}) \times \underline{\underline{I_{kn}}}\right)$$

Serial

The corresponding serial rule is:

$$\mathcal{R}_{k'}^{D,\dots} : MMM_{m,k,n}^{t,\delta} \rightarrow \left\{ \begin{array}{l} \left(\underline{\underline{\underline{I_{mn}}}} \otimes \underline{\underline{\underline{\underline{ACC_{tk'/k+\delta-1,k/k',tk'/k}}}}} \right) \circ \\ MMM_{m,k',n}^{tk'/k,\delta} \circ \\ \left(\left(\underline{\underline{\underline{I_{mk'}}}} \otimes \underline{\underline{\underline{\underline{SER_{k/k',0,tk'/k}}}}} \right) \times \right. \\ \left. \left(\underline{\underline{\underline{L_n^{nk'}}}} \cdot \left(\underline{\underline{\underline{I_{nk'}}}} \otimes \underline{\underline{\underline{\underline{SER_{k/k',0,tk'/k}}}}} \right) \cdot \underline{\underline{\underline{L_k^{kn}}}} \right) \right) \end{array} \right.$$

This rule serializes columns from the left input matrix, and rows from the right one. Those corresponding blocks are then multiplied together before being summed up to obtain the result.

3.3 Designs for MMM

3.3.1 Scalar Product

Before considering designs that compute a general MMM, let's first consider scalar products (i.e. $MMM_{1,k,1}$).

Adder tree

A scalar product can be performed by using only the rule $\mathcal{R}_1^{D,//}$. If we use it to perform a scalar product of two 4-dimentions vectors, we get the formula (after cleaning):

$$MMM_{1,4,1}^{1,0} \leftarrow S_4 \otimes MMM_{1,1,1}^{1,0}$$

This formula would be translated to the following Verilog code, where MUL is the module that multiplies $i1$ with $i2$ and that returns the result in $o1$:

```
...
    wire [15:0] var1;
    MUL MUL1(.i1(A_1_1),.i2(B_1_1),.o1(var1),
         .clk(clk),.rst(rst));
    wire [15:0] var2;
    MUL MUL2(.i1(A_1_2),.i2(B_2_1),.o1(var2),
         .clk(clk),.rst(rst));
    wire [15:0] var3;
    MUL MUL3(.i1(A_1_3),.i2(B_3_1),.o1(var3),
         .clk(clk),.rst(rst));
    wire [15:0] var4;
    MUL MUL4(.i1(A_1_4),.i2(B_4_1),.o1(var4),
         .clk(clk),.rst(rst));
    wire [15:0] var5;
    assign var5 = var1 + var2 + var3 + var4;
...
```

The way the additions of the last line are implemented depends on the synthetiser. Two scenarios may happen:

- The synthetiser can group one adder with one multiplier to form a MADD (to reduce the number of DSP slices used) (cf. Fig. 3.3.1). In this case, k DSP slices would be used, but the longest path would go through all k MADDs.

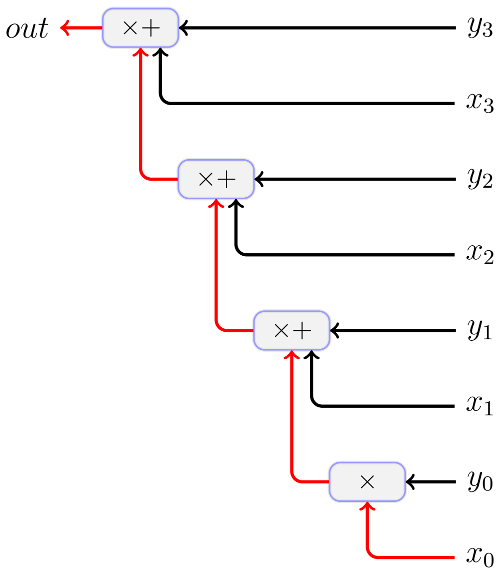

Figure 3.4: Design with MADDs

- The synthetiser can choose to produce an adder tree (cf. Fig. 3.3.1) to reduce the longest path. The drawback is that half of the multipliers cannot be grouped with an adder, resulting in an increased use of $k/2$ DSP slices. However, the longest path is better: one multiplier, one MADD, and $log_2(k) - 1$ adders.

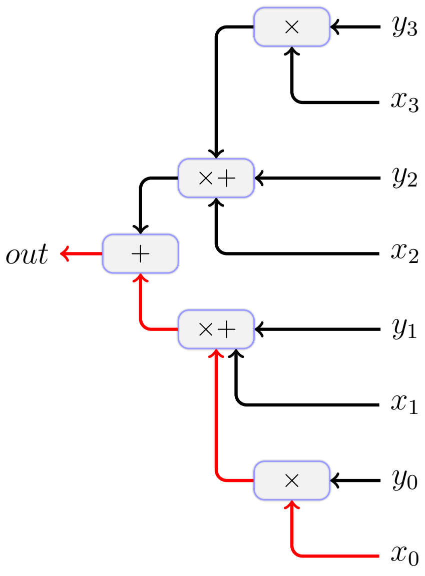

Figure 3.5: Design with an adder tree

To increase the frequency of operation, the adder tree could be made explicit and flip flops could be added to create a pipeline. However, in this case, the number of DSP slices used would still be high.

Cascaded MADDs

A correct way of improving the performance of a scalar product is to pipeline the first scenario (cf Fig. 3.3.1). To do that, we introduce two new operators, $\underline{\underline{\underline{CASC}}}$ and $\underline{\underline{\underline{CASCADD}}}$:

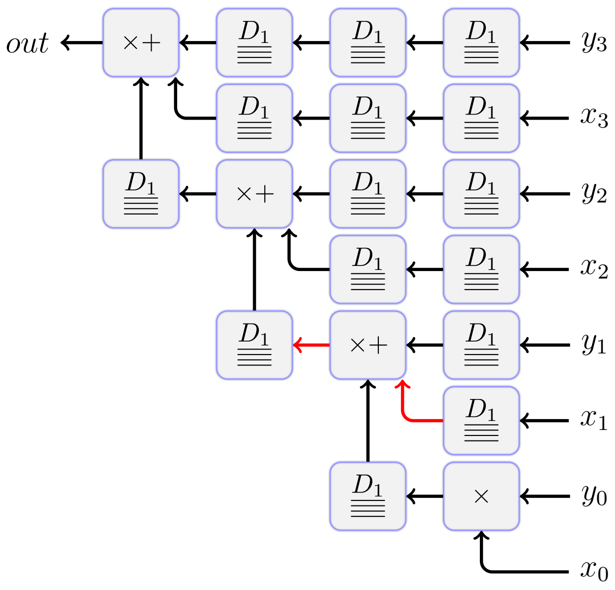

Figure 3.6: Design with cascaded MADDs

Cascader A cascader is an arity (1,1) linear operator that delays its inputs according to their position. A parameter δ indicates the number of cycles to delay the $i+1$-th input over the i-th:

$$\underline{\underline{\underline{CASC_{n,\delta}}}} = \underline{e_i} \odot \underline{e_{t+\delta\cdot i}} \odot \underline{e_t} \odot \underline{e_i}$$

The Verilog implementation uses only D-type flip-flops. A $\underline{\underline{\underline{CASC_{2,2}}}}$ will be implemented as follows:

```
reg [15:0] var1;
always @(posedge clk)
        var1 <= in1;
reg [15:0] out1;
always @(posedge clk)
        out1 <= var1;

reg [15:0] var3;
always @(posedge clk)
        var3 <= in2;
reg [15:0] var4;
always @(posedge clk)
```

```
        var4 <= var3;
reg [15:0] var5;
always @(posedge clk)
        var5 <= var4;
reg [15:0] out2;
always @(posedge clk)
        out2 <= var5;
```

Cascade adder The second operator we introduce is the cascade adder. It is an arity (1,1) linear operator that takes a n-dimentional vector and that outputs a scalar:

$$\underline{\underline{\underline{\underline{CASCADD_{n,\delta}}}}} = \underline{e_0} \odot \underline{e_t} \odot \underline{e_{t-\delta i}} \odot \underline{e_i}$$

The Verilog implementation uses flip-flops and adders. A $\underline{\underline{\underline{\underline{CASCADD_{2,2}}}}}$ is implemented:

```
reg [15:0] var1;
always @(posedge clk)
        var1 <= in1;
reg [15:0] var2;
always @(posedge clk)
        var2 <= var1;
wire [15:0] out;
assign out = var2 + in2;
```

Note that even if those two operators are sequential operators, they do not use any trigger. Therefore, no counter has to be produced.

Rule To use the two new operators, we introduce the rule:

$$\mathcal{R}^C : MMM^{1,\delta}_{1,k,1} \rightarrow \underline{\underline{\underline{\underline{CASCADD_{k,\delta/k}}}}} \circ \left(H_k \otimes MMM^{1,0}_{1,1,1}\right) \circ \left(\underline{\underline{\underline{\underline{CASC_{k,\delta/k}}}}} \times \underline{\underline{\underline{\underline{CASC_{k,\delta/k}}}}}\right)$$

This rule builds a scalar product with a gap of 1 cycle, and a variable latency of δ cycles.

Pipeline The DSP slices that perform the operations can be internally pipelined. It is possible to indicate to the synthesiser that we want such a behavior by placing flip-flops on the operator's output. With $\mathcal{R}^C$, this is exactly what happens when $\delta > 1$. For the adder tree, we introduce the rule:

$$\mathcal{R}^P : MMM^{1,\delta}_{1,1,1} \rightarrow \underline{\underline{\underline{\underline{D_\delta}}}} \circ MMM^{1,0}_{1,1,1}$$

We have two different ways to perform a scalar product (cf. Table 3.3).

3.3.2 General MMM

Parallel MMM

A general $MMM_{m,k,n}$ can be done by computing mn scalar products in parallel. This can be done by using $\mathcal{R}^{H,//}_1$, and $\mathcal{R}^T \circ \mathcal{R}^{H,//}_1 \circ \mathcal{R}^T$ before a scalar product rule.

This method produces the most performant MMM (with a gap of 1 cycle), but consumes a high number of DSP slices.

Name	Definition	Description
$\mathcal{R}^C$	$\mathcal{R}^C$	Cascaded adder
$\mathcal{R}^A$	$\mathcal{R}^P \circ \mathcal{R}_1^{D,//}$	Pipelined adder-tree

Table 3.3: Two families of scalar products

Family name	Kernel implemented	Definition
Eml	$MMM_{m,k,n}^{\frac{mkn}{xyz},p}$	$\mathcal{R}^A \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}^T \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{D,\cdots} \circ \mathcal{R}_x^{V,\cdots} \circ \mathcal{R}^T \circ \mathcal{R}_y^{V,\cdots}$
Clr	$MMM_{m,k,n}^{\frac{mkn}{xyz},pz}$	$\mathcal{R}^C \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}^T \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{D,\cdots} \circ \mathcal{R}_x^{V,\cdots} \circ \mathcal{R}^T \circ \mathcal{R}_y^{V,\cdots}$
Apo	$MMM_{m,k,n}^{\frac{mkn}{xyz},p}$	$\mathcal{R}^A \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}^T \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_x^{V,\cdots} \circ \mathcal{R}^T \circ \mathcal{R}_y^{V,\cdots} \circ \mathcal{R}_z^{D,\cdots}$
AG	$MMM_{m,k,n}^{\frac{mkn}{xyz},pz}$	$\mathcal{R}^C \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}^T \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_x^{V,\cdots} \circ \mathcal{R}^T \circ \mathcal{R}_y^{V,\cdots} \circ \mathcal{R}_z^{D,\cdots}$

Table 3.4: Design families for MMM

Serialisation

To compute large MMMs with a limited amount of DSP slices, a good option is to block it into several sub-MMM, and to use the same circuitry to compute these sub-MMM one after the other. This can be done by using $\mathcal{R}_x^{V,\cdots}$, $\mathcal{R}_z^{D,\cdots}$ and $\mathcal{R}^T$ before the rules that produce a parallel MMM.

Design families

In the next chapter, we will explore four design families (see Table 3.4). Each of these families takes 4 parameters, x, y, z and p. x, y and z are used to serialy block the MMM into a $MMM_{x,y,z}$, and therefore must be divisors of m,k and n, respectively. The parameter p is used to specify a pipeline depth. Also it can take any integer value, we will only explore $\mathbb{N}^4$.

The difference between those families are the following:

- Eml and Apo use an adder-tree based scalar product, while Clr and AG use a cascaded scalar product.
- Eml and Clr perform the depth serialisation after the vertical and horizontal serialisation, Apo and AG after.

The number of cycles to wait before feeding new matrices, the*gap*, and the additional latency required by each design is indicated in the "Kernel implemented" column in Table 3.4.

4 Results

4.1 Experimental Setup

The design space we generated was synthesised for a Xilinx xc6vlx75t-ff484-1. This FPGA contains 11640 *slices*, that can be used for general purpose logic, and 288 *DSP48e1*, that contain hard-wired circuitry to perform operations that would require a lot of slices otherwise like additions, multiplications, or both.

Therefore, we used the Xilinx toolchain using the following script:

```
echo 'run -move_first_stage no -move_last_stage ' > mmm.xst
echo 'no -ifn mmm.v -ifmt Verilog ' >> mmm.xst
echo '-ofn mmm.ngc -iobuf no -ofmt ' >> mmm.xst
echo 'NGC -p xc6vlx75t-ff484-1 -top wrapper' >> mmm.xst
xst -ifn mmm.xst
map -u -timing -ol high -xe n -t 1 -xt 0 mmm.ngd
par mmm out -ol high -xe n
trce -a out
```

4.2 Synthesis architecture

The synthesis is the most time consuming stage of the process (a $MMM_{16,16,16}$ can take up to 15 hours on an Intel Xeon(R) X5680 @3.33GHz with 141GB of RAM. Therefore, we used a distributed architecture composed of:

- A server selects a design to test, generates the corresponding verilog code, and sends it to the clients using a web server. It also runs a database where it stores the results sent by the clients. A web interface is also provided to allow the customer to pick his design.
- Clients download the Verilog code from the server, run the synthesis, and upload the results (maximum frequency, number of slices used, logs) to the server.

4.3 Scalar Products

We first focus on scalar products performances. Fig. 4.1 shows the maximum frequency of a scalar product as a function of the input width (k). Here, no serialisation is applied; only the rules $\mathcal{R}^A$ and $\mathcal{R}^C$ are used with different level of pipeline.

As expected, the maximum frequency of the adder tree ($\mathcal{R}^A$) decreases with the input size. With no pipeline, this value decreases from 180MHz ($k = 2$) to 12MHz ($k = 48$). With a 3-levels pipeline, the maximum frequency is 8 times higher.

The cascaded MADDs shows a maximal frequency nearly constant until $k = 32$ (the longuest path is constant in this case). Then, it starts decreasing, probably because

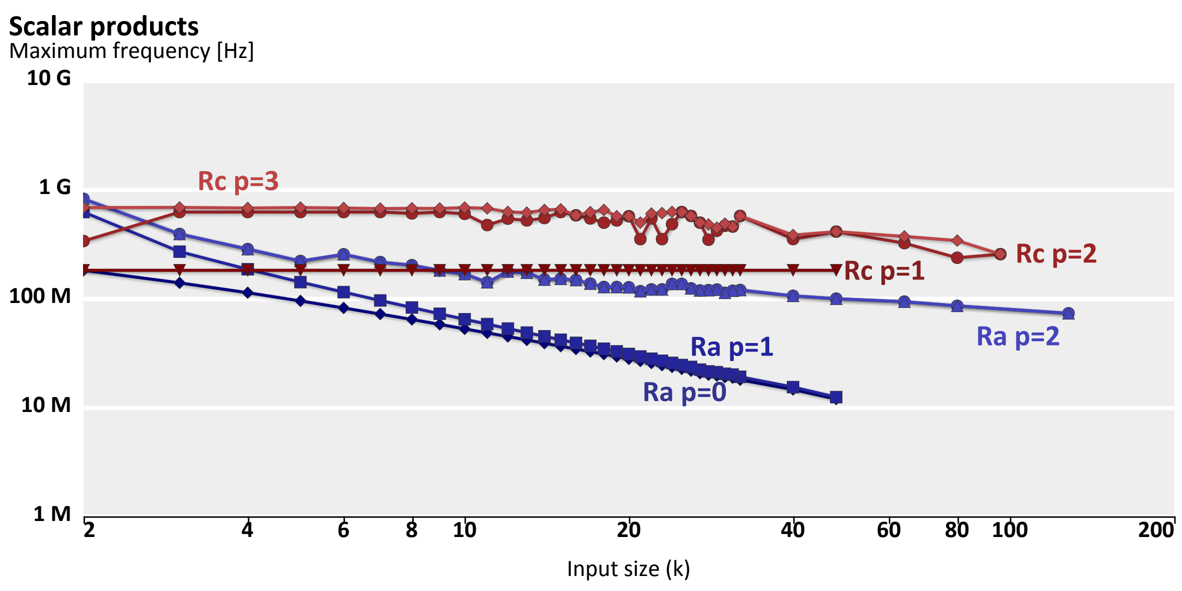

Figure 4.1: Performance of scalar products

of placement difficulties. Once again, the pipeline level plays an important role. A 1-pipelined cascaded scalar product runs at 180MHz, while a 3-pipelined one runs at 672MHz.

4.4 Influence of the complexity

The maximum frequency over all designs obtained for MMM of square matrices is shown on Fig. 4.2.

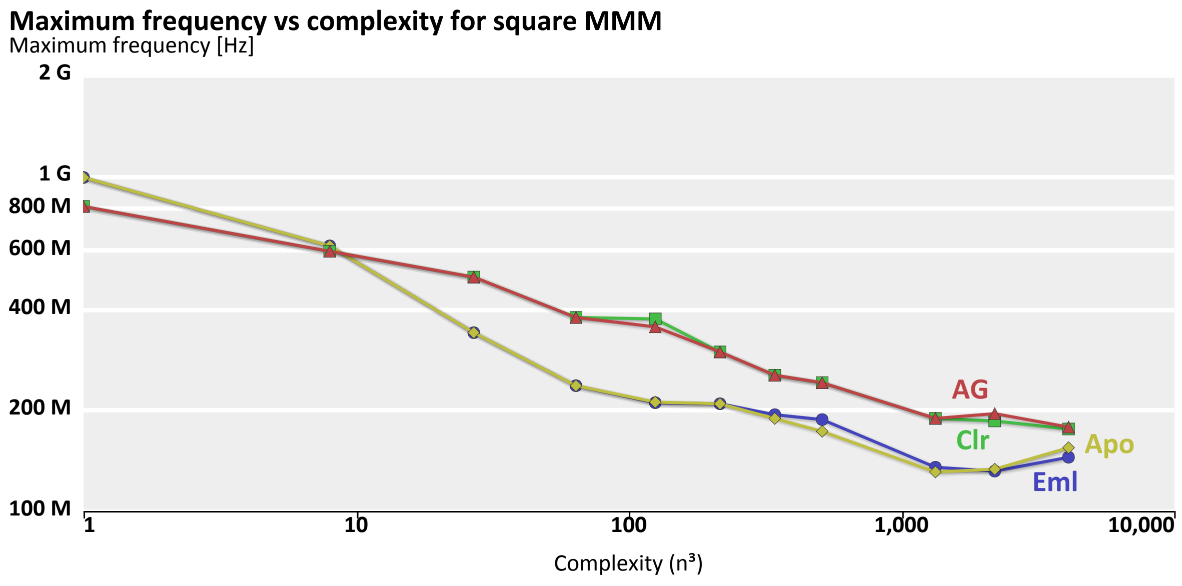

Figure 4.2: Frequency vs complexity

The maximum frequency obtained decreases with the complexity, even for designs which scalar product had a steady frequency until $k = 32$. However, it is noticeable that the designs that use the cascaded MADDs (Clr, AG) have a higher frequency. This lends weight to the idea that it is the placement and route step that limits the highest achievable frequency.

Fig.4.3 shows the maximum performance obtained over all designs for a MMM of square matrices.

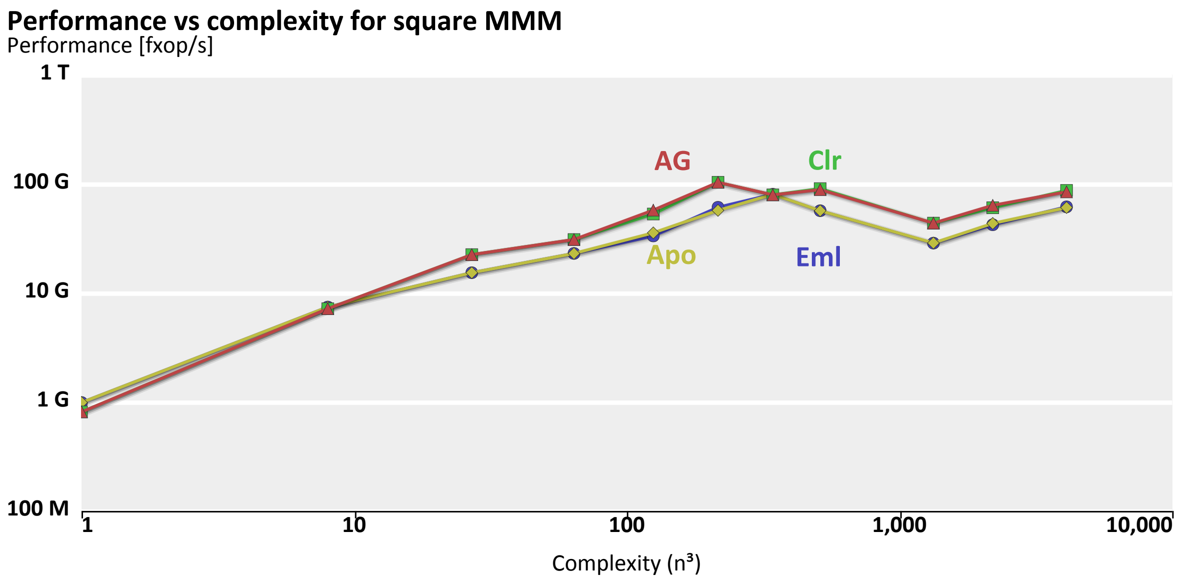

Figure 4.3: Performance vs complexity

The maximum performance increases until $n = 6$, which corresponds to a complexity of 216 multiplications and additions. Until this point, all of the DSP48e1 that are available on the FPGA are not used. Therefore, the more the complexity increases, the more DSP slices are used, and the better the performance is. The designs that use the cascaded MADDs have better performance, probably because of a higher frequency.

4.5 MMM of square matrices

In this section, we show the design space generated for some MMM problems. For a $MMM_{m,k,n}$, each design family provides $4 \times D(m) \times D(k) \times D(n)$, where $D(x)$ is the number of divisors of x. Therefore the following formula gives the number of designs generated:

$$16 \times D(m) \times D(k) \times D(n)$$

The interesting characteristics of a design are its maximal frequency, its performance (the number of MMM it can compute in a second), the number of DSPs and the number of slices it uses.

Some design are better than others: they have a better performance than every designs that use less slices and less DSPs. Those designs form the Pareto set of the problem. If a design A is a Pareto optimum, we have for any design B:

$$\text{Performance}(B) > \text{Performance}(A) \Rightarrow \#\text{Slices}(B) > \#\text{Slices}(A) \text{ or } \#\text{DSP}(B) > \#\text{DSP}(A)$$

4.5.1 2x2

Fig. 4.4 and 4.5 show the generated design space for a kernel that multiplies two 2×2 matrices. The Pareto points (black circle) represent the designs that are the most performant for a given number of slices and DSPs.

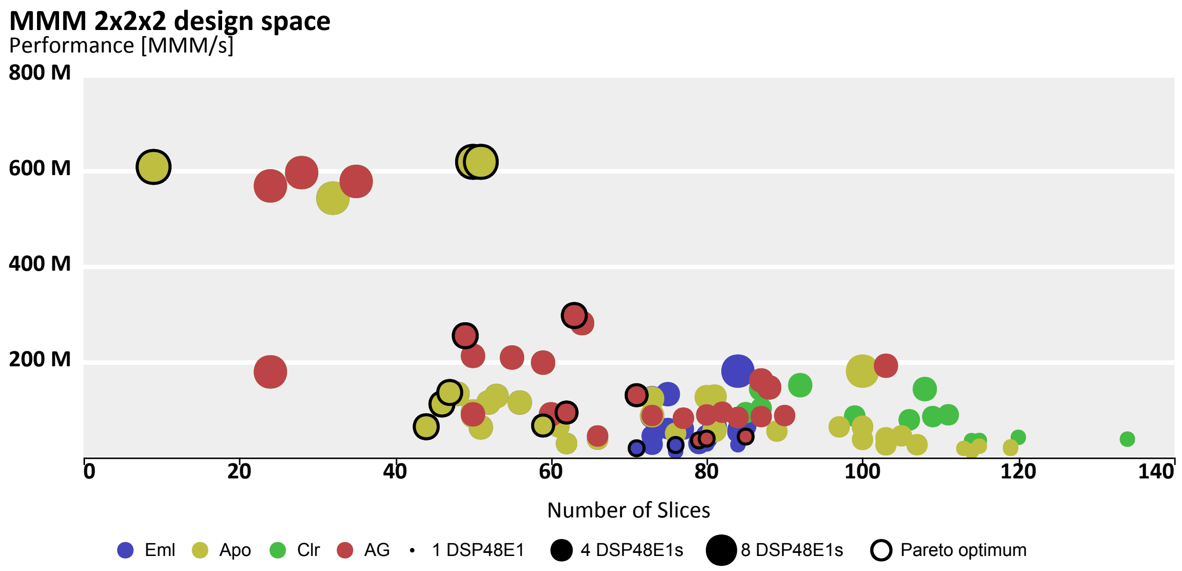

Figure 4.4: MMM 2x2x2 performances

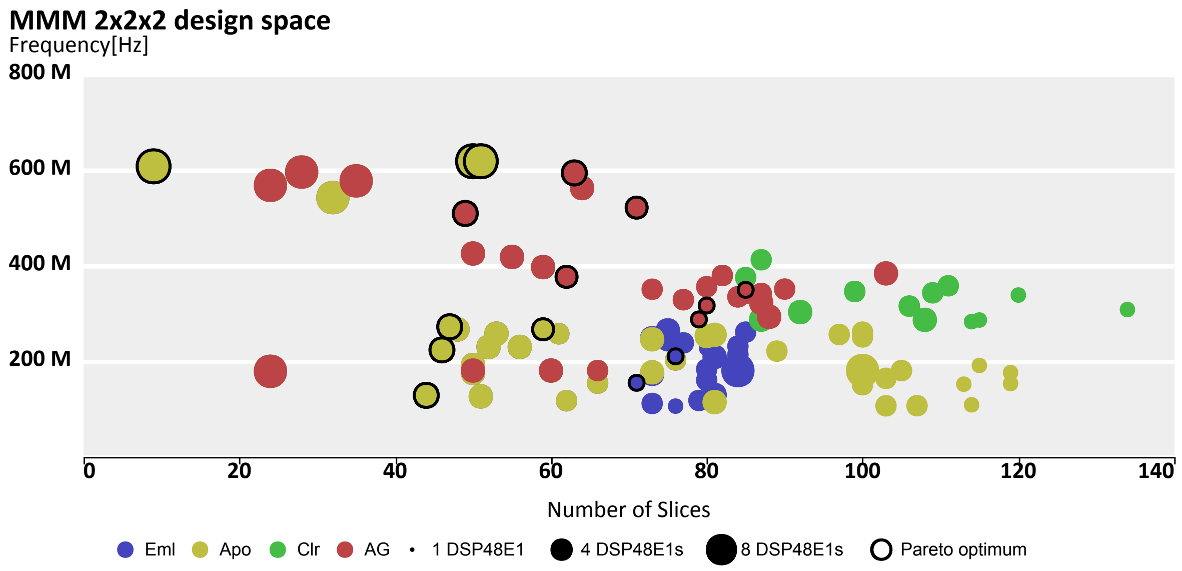

Figure 4.5: MMM 2x2x2 maximum frequencies

The frequencies vary between 150 and 600MHz. Some designs output matrices at the same speed: the ones that use 8 DSPs. In this case, the computation is fully done in parallel, and the gap of the kernel is 1. For the others, the frequency at wich matrices are output is divided by 2 or by 4, depending on the serilization used.

As an $MMM_{2,2,2}$ requires only 2-inputs scalar products, the difference of performance between the families that use different scalar products is not important: every design family is represented in the Pareto set.

4.5.2 4x4

Fig. 4.6 and 4.9 show the generated design space for a kernel that multiplies two 4 × 4 matrices.

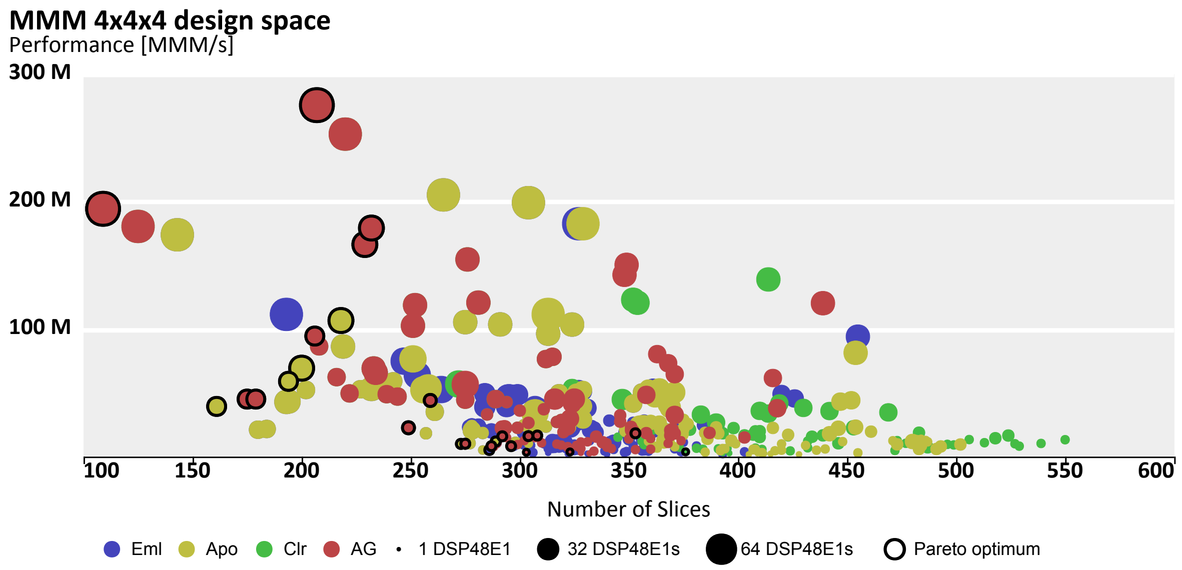

Figure 4.6: MMM 4x4x4 design space

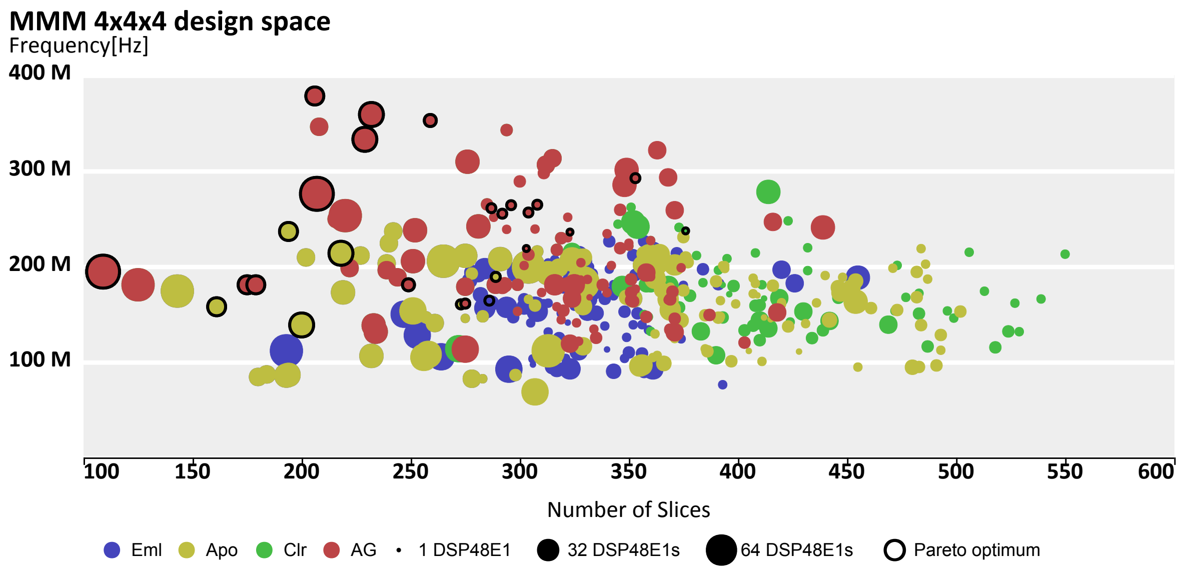

Figure 4.7: MMM 4x4x4 maximum frequencies

The frequency range is lower, they vary between 75 and 380MHz. The difference between the different scalar products begin to appear; most of the Pareto points are from the

AG and Clr families. The other families are only represented in highly serialized designs (with small scalar products).

4.5.3 8x8

Fig. 4.8 and 4.9 show the generated design space for a kernel that multiplies two 8×8 matrices.

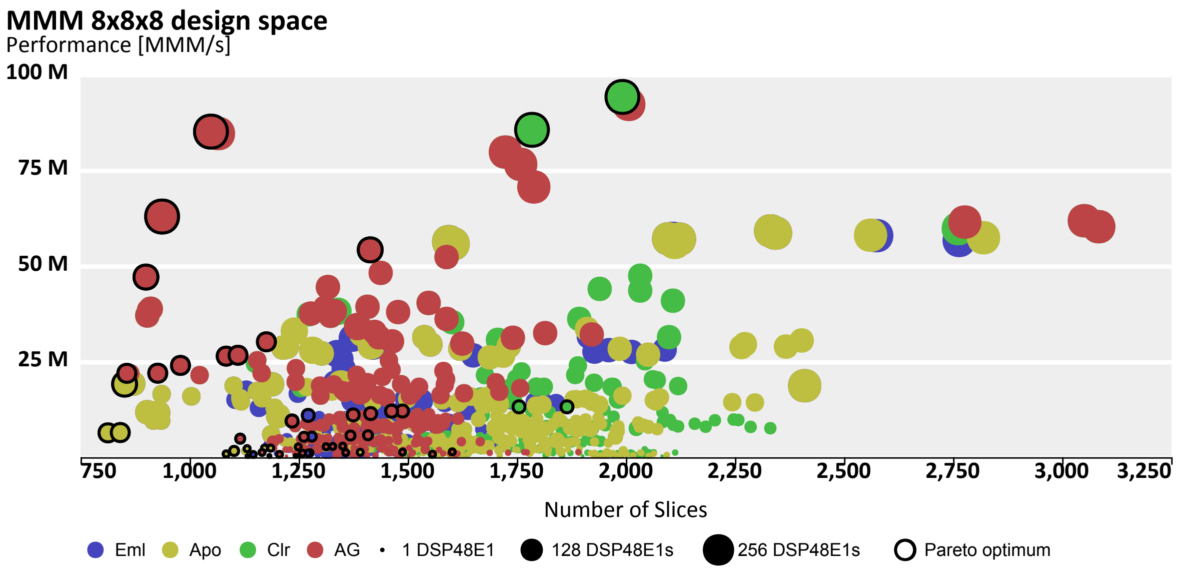

Figure 4.8: MMM 8x8x8 design space

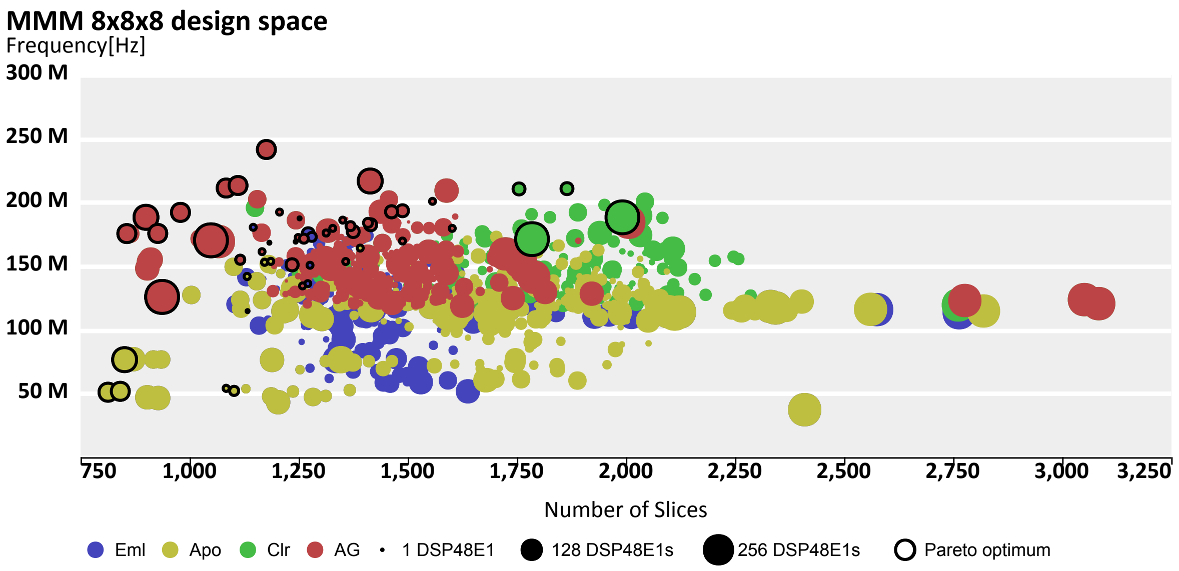

Figure 4.9: MMM 8x8x8 maximum frequencies

The frequency range is once again lower, between 45 and 240 MHz. Except for highly serialised designs, the Pareto points come from the families that use the $\mathcal{R}^C$ scalar product.

4.5.4 16x16

Fig. 4.11 and 4.10 show the generated design space for a kernel that multiplies two 16×16 matrices. Here, designs with low level pipeline (< 2) are not displayed.

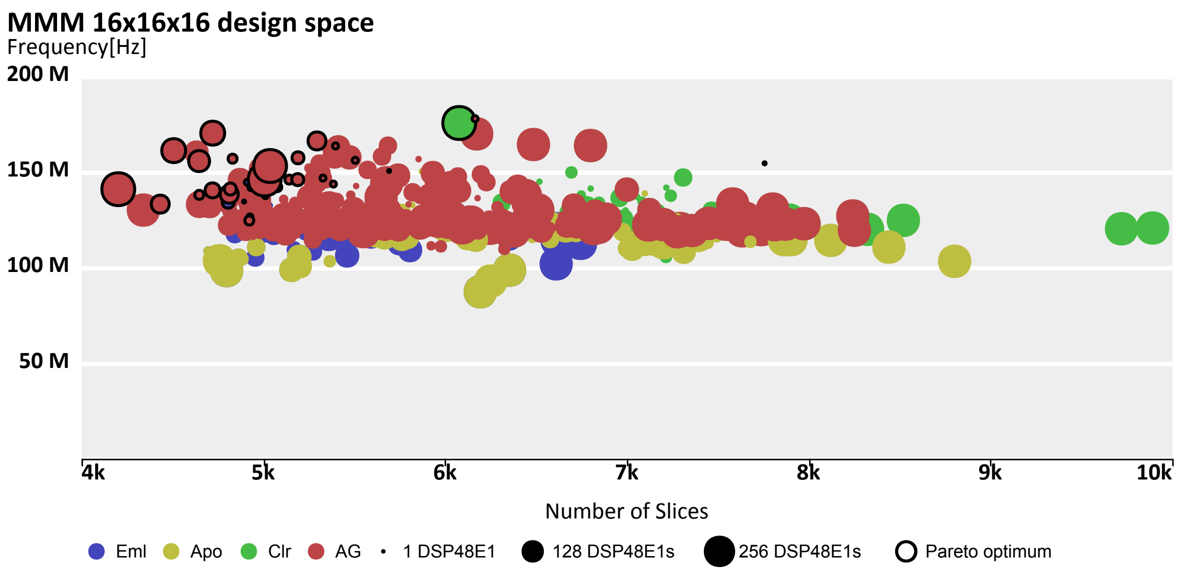

Figure 4.10: MMM 16x16x16 maximum frequencies

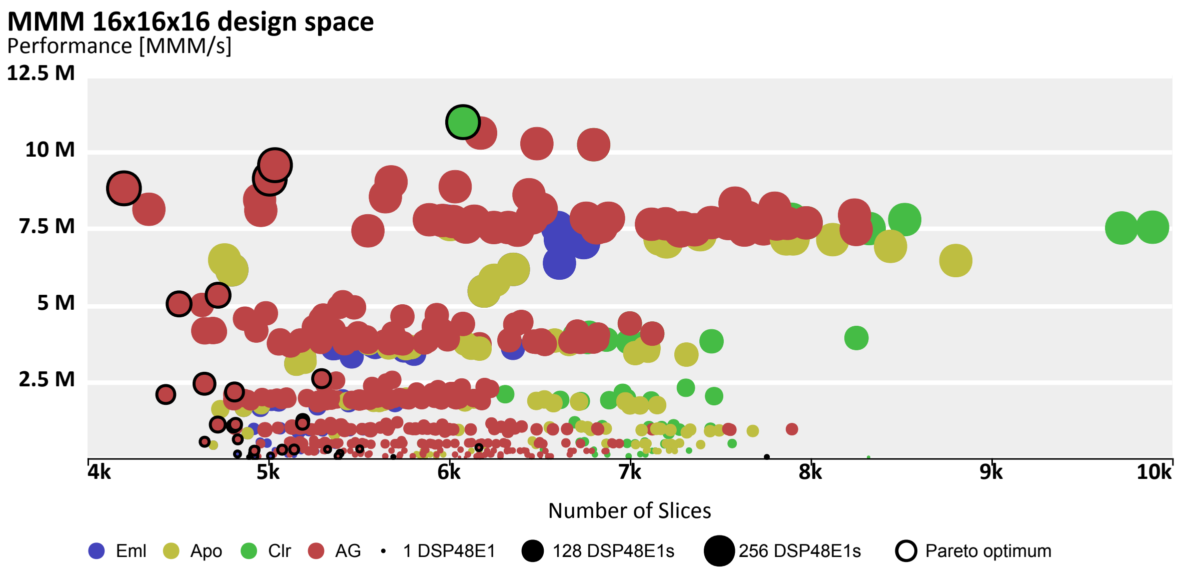

Figure 4.11: MMM 16x16x16 design space

Once again, Clr and AG are the best design families. As we removed low pipelined designs, the frequencies are mostly grouped around 120MHz. There is a quantization of the performances: as the frequencies become uniform, the only value that pilots the performance is the gap, an integer.

4.6 Non-Square Matrices

Our generator is able to generate non-square and/or non-power of 2 matrices. Fig. 4.12 shows the design space for an MMM that will multiply a 4×5 matrix with a 5×7 matrix.

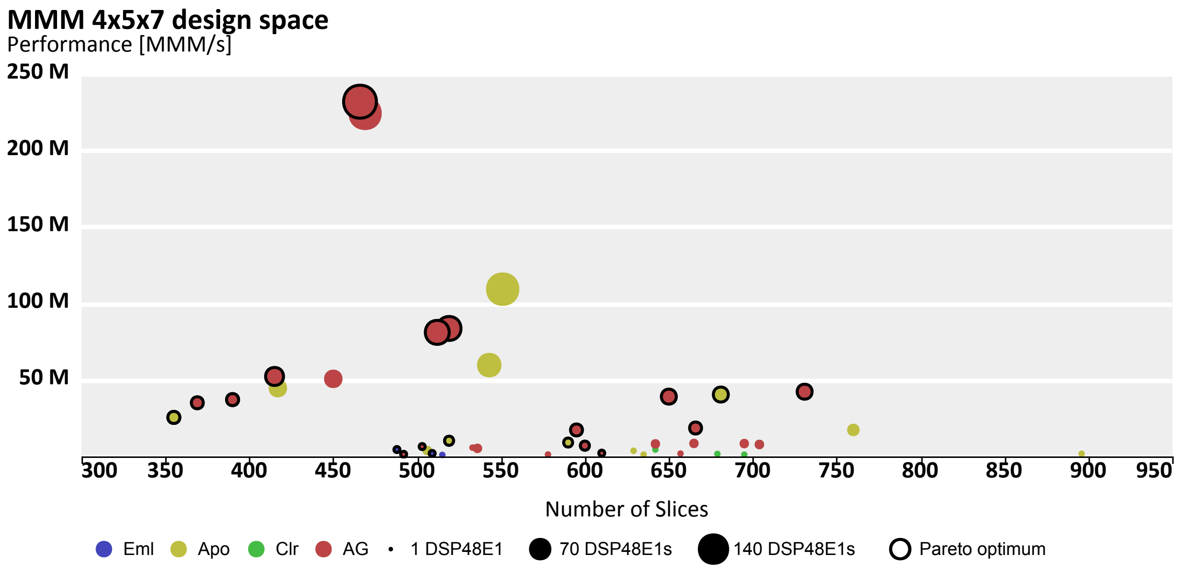

Figure 4.12: Performance of an MMM 4x5x7

As 5 and 7 are prime numbers, the number of designs is smaller than for other designs.

4.7 Limited Frequency

In the case where the MMM design shares its clock with another circuitry, it may not be possible for it to run at its full frequency. Fig.4.13 shows the case of an $MMM_{8,8,8}$ limited to 100MHz. The quantization of the performance appears once again. In this case, the influence of the scalar product used is inversed. Cascaded products with high pipeline tend to use more logic slices than the other designs, with no benefit since the maximum frequency does not count. Therefore, the Pareto set contains designs from all families.

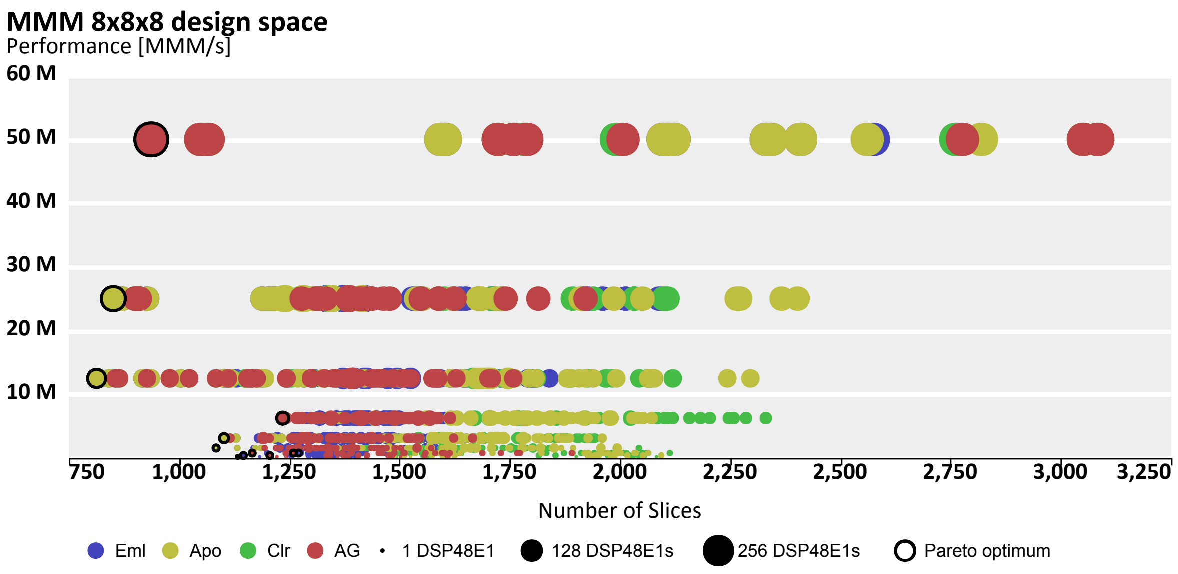

Figure 4.13: MMM 8x8x8 limited to 100MHz

5 Conclusion

In this thesis, we presented a way to automatically generate designs that perform MMM on an FPGA. The produced design space is then synthesised to collect the characteristics of each design. Finally, given a set of constraints, a user can choose the most appropriate design to perform an MMM operation.

In this document, we considered only the case of MMM, and Verilog implementations. It is possible to use the formalism described here for other kernels and platforms where time and/or reuse plays an important role. However, the following improvements can be considered: A real operator is always periodic, and this doesn't appear in the definitions we used here. It would be interesting to quotient the space of operators over the equivalence relation "has the same gap". It would then be possible to define a Kronecker product that takes a sequential operator as a left term and that does what we would expect it to do: describe how the right operator should be (re)used both in time and space (this would avoid the dissimetry we currently have between the "parallel rules" and the "serial rules"). This extended Kronecker product could also be used to describe easily assymetric algorithms (in the case a GPGPU is used simultaneously with a CPU for instance).

Another enhancement for our generator would be to handle correctly the case where the input matrices and/or the output matrix are streamed. In fact, we supposed that both of the input matrices and the output matrix were completly provided in one cycle. It is possible to handle streaming by using specific operators in the starting formula:

$$\underline{\underline{\underline{I_2}}} \otimes \underline{\underline{\underline{\underline{\underline{SER_{2,1,1}}}}}} \circ MMM^{1,0}_{2,2,2} \circ \left(\left(\underline{\underline{\underline{I_2}}} \otimes \underline{\underline{\underline{\underline{\underline{DESER_{2,0,1}}}}}} \right) \times \left(\underline{\underline{\underline{I_2}}} \otimes \underline{\underline{\underline{\underline{\underline{DESER_{2,0,1}}}}}} \right) \right)$$

If the above formula is used as a starting kernel, it will generate a streamed $MMM_{2,2,2}$ with a streaming factor of 2. However, this solution is inelegant in the sense that the generated kernel will wait to have the whole input matrices to begin the computation, and will begin the streaming of the output once the computation is done. A correct way to handle it would be to use a systematic way to implement streaming permutations with registers, as described in [8].

Bibliography

[1] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How to write fast numerical code: A small introduction. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer, 2008.

[2] Frédéric de Mesmay, Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Automatic generation of adaptive libraries for matrix-multiplication. Parallel Matrix Algorithms and Applications (PMAA), 2008. Presentation (Abstract reviewed).

[3] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. 2005.

[4] Franz Franchetti, Frédéric Mesmay, Daniel Mcfarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 385–409, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] PRASANNA Viktor K. JANG Ju-Wook, CHOI Seonil. Area and time efficient implementations of matrix multiplication on fpgas. *IEEE, New York, NY, ETATS-UNIS (2002).*

[6] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012.

[7] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[8] Markus Püschel, Peter A. Milder, and James C. Hoe. Permuting streaming data using rams. *Journal of the ACM*, 56(2):10:1–10:34, 2009.

[9] Patrick Le Tallec. *Mécanique des milieux continus*. École polytechnique, 2008.

[10] Xilinx. Logicore ip axi linear algebra toolkit data sheet. Technical report, January 2011.

[11] Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on fpgas. *Parallel and Distributed Processing Symposium, International*, 1:92a, 2004.

[12] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 63–74, New York, NY, USA, 2005. ACM.