Master Thesis

Automatic Generation of Hardware Designs for Matrix-Matrix Multiplication

Author(s):
Serre, François

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007636137

Rights / License:
In Copyright - Non-Commercial Use Permitted
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

1 Introduction .................................................. 1
   1.1 Motivation ............................................... 1
   1.2 Previous work ........................................... 1
      1.2.1 MMM Hardware Implementations .................... 1
      1.2.2 Languages to describe an Algorithm ............... 1
   1.3 Thesis Overview ........................................ 2
   1.4 Notations ................................................. 3
   1.5 Organization ............................................. 3

2 Operator Language for Schedules ............................ 5
   2.1 A reminder on tensors ................................... 5
      2.1.1 Tensor ............................................... 5
      2.1.2 Canonical basis ..................................... 6
      2.1.3 Tensor Product ...................................... 6
      2.1.4 Simply Contracted Product ......................... 7
      2.1.5 Doubly Contracted Product ......................... 7
   2.2 Schedules ................................................ 8
   2.3 Operators ................................................. 8
      2.3.1 Combinational Operators ........................... 9
      2.3.2 Sequential Operators ............................... 10
      2.3.3 Composition ......................................... 11
      2.3.4 Kronecker Product ................................ 12
   2.4 Rules .................................................... 13

3 Matrix-Matrix Multiplication in Verilog ..................... 15
   3.1 Formula to Verilog ...................................... 15
      3.1.1 Composition ......................................... 15
      3.1.2 Combinational Operators ........................... 15
      3.1.3 Sequential Operators ............................... 16
      3.1.4 Kronecker Products ................................ 19
      3.1.5 New Cleaning Rules ................................ 20
   3.2 Specific Rules for MMM ................................. 20
      3.2.1 Vertical and Horizontal Blocking .................. 20
      3.2.2 Depth Blocking ...................................... 22
   3.3 Designs for MMM .......................................... 23
      3.3.1 Scalar Product ..................................... 23
      3.3.2 General MMM ........................................ 26

4 Results ....................................................... 29
   4.1 Experimental Setup ...................................... 29
   4.2 Synthesis architecture ................................. 29
# List of Figures

2.1 An operator working on a dataflow ........................................ 8  
2.2 Implementation of an adder ........................................... 9  
2.3 Implementation of a multiplexer ....................................... 11  
2.4 Composition of two Operators .......................................... 11  

3.1 Implementation of a D-type flip-flop ................................. 16  
3.2 Implementation of an accumulator ..................................... 18  
3.3 The different ways to block an MMM ................................. 21  
3.4 Design with MADDs ..................................................... 24  
3.5 Design with an adder tree ............................................. 24  
3.6 Design with cascaded MADDs .......................................... 25  

4.1 Performance of scalar products ........................................ 30  
4.2 Frequency vs complexity ............................................... 30  
4.3 Performance vs complexity ............................................. 31  
4.4 MMM 2x2x2 performances ............................................. 32  
4.5 MMM 2x2x2 maximum frequencies ................................... 32  
4.6 MMM 4x4x4 design space ............................................. 33  
4.7 MMM 4x4x4 maximum frequencies ................................... 33  
4.8 MMM 8x8x8 design space ............................................. 34  
4.9 MMM 8x8x8 maximum frequencies ................................... 34  
4.10 MMM 16x16x16 maximum frequencies ............................... 35  
4.11 MMM 16x16x16 design space ........................................ 35  
4.12 Performance of an MMM 4x5x7 ..................................... 36  
4.13 MMM 8x8x8 limited to 100MHz ..................................... 37
List of Tables

2.1 Examples of order $n$ tensors ........................................... 5
2.2 Cleaning rules ................................................................. 14

3.1 Possible left term for a Kronecker product in a terminated formula ... 19
3.2 Additional cleaning rules ................................................... 21
3.3 Two families of scalar products .......................................... 27
3.4 Design families for MMM .................................................. 27
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 cases, 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].


In its Linear Algebra Toolkit [10], Xilinx developed a tool able to generate a MMM for FPGAs. The user can set the dimensions 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
Introduction

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 characteristics. 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 constraints on a web interface.
1.4 Notations

In this thesis, the tensor product (produces a $p+q$ order tensor) is noted $\odot$ to differentiate 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 tensors that are used in this document.

\(^1\)This is a very cheap definition, but we don’t need more

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

Table 2.1: Examples of order $n$ tensors
2.1.2 Canonical basis

The notation $e_{i_1} \circ e_{i_2} \circ ... \circ e_{i_n}$ 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 $(e_{i_1} \circ ... \circ e_{i_n})_{i_1,...,i_n}$ is an orthonormal basis of the set of order $n$ tensors. Therefore, every tensor $T$ can be rewritten as:

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

Examples

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

2.1.3 Tensor Product

Definition

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

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

Examples

- The tensor product of two vectors $\mathbf{u} = u_i e_i$ and $\mathbf{v} = v_j e_j$ is the matrix:
  $$\mathbf{u} \circ \mathbf{v} = (u_i \times v_j) e_i \circ e_j = \mathbf{u} \cdot \mathbf{v}$$
- The tensor product of two matrices $M = m_{i,j} e_i \circ e_j$ and $N = n_{k,l} e_k \circ e_l$ is the order 4 tensor:
  $$M \circ N = (m_{i,j} \times n_{k,l}) e_i \circ e_j \circ e_k \circ e_l$$
- 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 posteriori our notation $e_{i_1} \circ ... \circ e_{i_n}$ 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 A reminder on tensors

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) $A \cdot B$ is the $p + q - 2$-order tensor such that:

$$A \cdot B = (a_{i_1,\ldots,i_{p-1},k} \times b_{k,i_{p+2},\ldots,i_{p+q}})e_{i_1} \otimes \cdots \otimes e_{i_{p-1}} \otimes e_{i_{p+2}} \otimes \cdots \otimes e_{i_{p+q}}$$

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 $u = u_i e_i$ and $v = v_j e_j$ is the scalar:
  $$u \cdot v = u_i v_i$$
  In other words, the contracted product of two vectors is the canonical scalar product of $\mathbb{R}^k$.

- The contracted product of a matrix $M = m_{i,j} e_i \otimes e_j$ and a vector $u = u_k e_k$ is the vector:
  $$M \cdot u = m_{i,j} u_j e_i$$
  This is the classical Matrix-Vector multiplication.

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

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

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!
2 Operator Language for Schedules

Examples

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

$$M : N = m_{i,j}n_{j,i} = \text{Tr}(M \cdot N)$$

If $M$ (and $N$) is square, then $M : N$ defines the classical scalar product on $M_k(\mathbb{R})$

- If $M$ is square, and if $I$ is the identity matrix ($I = e_i \odot e_i$), we have:

$$M : I = m_{i,j} \delta^i_j = m_{i,i} = \text{Tr}(M)$$

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

$$C : M = c_{i,j,k,l} m_{i,j} e_k \odot e_l$$

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 $v_i e_i$. $v_i$ 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 $s = s_{i,t} e_i \odot e_t$ $s_{i,t}$ 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 Operators

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 $M$ (the same as in the OL paper). The recipe to apply it on a schedule $s$ is the same as for a vector; a simply contracted product:

$$ r = M \cdot s $$

**Examples**

- The $N \times N$ matrix

$$ DFT_N = e^{-\frac{2\pi i}{N} (j-1)(k-1)} e_{j} \odot e_{k} $$

is the discrete Fourier transform. Applying it on a $N \times T$ matrix will perform $T$ Fourier transforms (1 per cycle).

- We note:

$$ ADD = \begin{bmatrix} 1 & 1 \end{bmatrix} $$

This typically describes an adder (adds two numbers every cycle, cf. Fig. 2.2).

![Figure 2.2: Implementation of $ADD$](image)

**Other Combinational Operators**

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

$$ B \left( s_1, s_2 \right) = B \left( s_1 e_t, s_2 e_t \right) \odot e_t $$
Examples

- We note $MMM_{m,k,n}$ the matrix multiplication that consumes two matrices and produces one:
  
  $$MMM_{m,k,n} : \mathbb{C}^{mk} \times \mathbb{C}^{kn} \rightarrow \mathbb{C}^{mn}; (A, B) \mapsto AB$$

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

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

The meaning of $t_{i',t,i}$ 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 $D_c$ the operator:

  $$D_c = e_{i} \odot e_{t+c} \odot e_{t} \odot e_{i}$$

  Let’s apply $D_c$ on a schedule:

  $$D_c : s = s_{i,t} e_{i} \odot e_{t+c}$$

  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 $[0; k - 1]^N$. Then, we denote:

  $$MUX_u = e_{0} \odot e_{t} \odot e_{t} \odot e_{u(t)}$$

  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:

  $$T_h = e_{i} \odot e_{h \times t} \odot e_{h \times t} \odot e_{i}$$

  At cycle $t$, this operator returns its input if $t$ is a multiple of $h$, and zeroes otherwise.
2.3 Operators

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).

Linear operator with arity \((1,1)\) can easily be composed using the two kinds of contracted products.
2 Operator Language for Schedules

**Two Linear Combinational Operators**

As the simply contracted product is associative, we have:

\[ A \cdot (B \cdot s) = (A \cdot B) \cdot s \]

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:

\[ \underline{A} : (\underline{B} : s) = (\underline{A} : B) : s \]

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:

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

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

\[ P_{l,h} = D_l : T_h \]

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

\[ P_{4,2} \cdot \underline{DFT}_N \]

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:

\[ A \otimes B = [a_{i,j}B], A = a_{i,j}e_i \otimes e_j \]
2.4 Rules

In the case where \( A = I_n \), we have:

\[
I_n \otimes B = \begin{bmatrix} B \\ \vdots \\ B \end{bmatrix}
\]

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 \( A : \mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}^r \) and any sequential operator \( B : \mathcal{M}_{m,T}(\mathbb{R}) \times \mathcal{M}_{n,T}(\mathbb{R}) \rightarrow \mathcal{M}_{k,T'}(\mathbb{R}) \):

\[
(A \otimes B)(x, y) = A(e_i, e_j) \otimes B \left( (e_0 \circ e_i) \otimes I_m \cdot x, (e_0 \circ e_j) \otimes I_n \cdot y \right)
\]

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 \( R \) 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:

\[
MMM_{0,1,1}^{0,1,1} \rightarrow D_1 \circ MUL
\]

replaces the first occurrence of \( MMM_{1,1,1}^{0,1,1} \) in a formula by the expanded version \( D_1 \circ MUL \).

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

\[
I_i \circ Expr \rightarrow Expr
\]

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 \( CR \) the set of cleaning rules applied in sequence.

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

\[
(I_1 \otimes Expr \rightarrow Expr)^\infty
\]

will remove all Kronecker products where the left term is \( I_1 \) 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, \( CR^\infty \) is applied on the resulting formula.
First cleaning rule set

Removal of useless kronecker products:

\[ I_i \otimes Expr \rightarrow Expr \]
\[ I_i \otimes I_j \rightarrow I_{ij} \]

Removal of useless identity operators:

\[ I_i \circ Expr \rightarrow Expr \]
\[ Expr \circ I_i \rightarrow Expr \]

\[ Expr \circ \left( I_i \times I_j \right) \rightarrow Expr \]
\[ \left( I_i \times I_j \right) \circ Expr \rightarrow Expr \]

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:

```verilog
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 $L_{mn}^{mn}$ is the arity (1,1) permutation that transposes a $m \times n$ matrix (seen as a linearized vector). Therefore:

$$L_{mn}^{mn} \cdot e_{i \times n+j} = e_{j \times n+i}$$
Swap  The swap operator, noted $SWAP$ is the arity $(2,2)$ operator that simply exchanges its inputs:

$$SWAP(x,y) = (y,x)$$

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:

```verilog
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:

$$D_1 = e_i \oplus e_{i+1} \oplus e_i \oplus e_i$$

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 $D_1$](image)

The corresponding Verilog code is the following:

```verilog
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 resources. 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 synthesiser 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 \textit{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.

\begin{verbatim}
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);
\end{verbatim}

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 \(\varphi\) modulo \(T\), and holds this value at the output until it gets a new one.

\[
MEM_{\varphi,T} = \sum_{0 \leq i < T} e_0 \odot e_{i+1} \odot e_i \odot e_0 \\
\]  

Using a trigger \(t\) that fires at cycle \(\varphi\) modulo \(T\), the Verilog implementation is straightforward:

\begin{verbatim}
reg [15:0] out;
wire [15:0] nextvalue;
assign nextvalue = t ? in : out;
always @(posedge clk)
  out <= nextvalue;
\end{verbatim}

**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 \(\delta\) specifies the number of cycles to wait between every additions. The other cycles are simply skipped:

\[
ACC_{\varphi,T,\delta} = \sum_{0 \leq i < T} e_{1} \odot e_{i+T\delta} \odot e_{i+\delta} \odot e_{1} \\
\]

The Verilog implementation uses two triggers: \(t0\), that fires every cycle \(\varphi\) modulo \(\delta\), and \(t1\) every cycle \(\varphi\) modulo \(T\delta\).
Matrix-Matrix Multiplication in Verilog

Figure 3.2: Implementation of 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 dimension $n$ vector as an input, and has a scalar output. It stores its input at cycle $\phi$ modulo $n \cdot \delta$, and outputs its components one after the other during $\delta$ cycles:

$$SER_{n,\phi,\delta} = \sum_{0 \leq i < n} \sum_{0 \leq j < \delta \mod n} e_1 \odot e_{t+i\delta+j} \odot e_t \odot e_i$$

The serializer is implemented using a shift register. Once again, two triggers are used in the Verilog implementation: $t_1$ to wait until the next $\delta$ cycle, and $t_2$ 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 $n$-dimensional vector that contains these values.
### 3.1 Formula to Verilog

#### Operator Name Definition

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_n$</td>
<td>Identity</td>
<td>$\mathbb{R}^n \to \mathbb{R}^n; x \mapsto x$</td>
</tr>
<tr>
<td>$H_n$</td>
<td>Hadamard product</td>
<td>$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n; (x, y) \mapsto x_i y_i e_i$</td>
</tr>
<tr>
<td>$S_n$</td>
<td>Scalar product</td>
<td>$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}; (x, y) \mapsto x_i y_i$</td>
</tr>
<tr>
<td>$K_{m,n}$</td>
<td>Kronecker product</td>
<td>$\mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{mn}; (x, y) \mapsto x_i y_{ij}$</td>
</tr>
</tbody>
</table>

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

As the previous operators, a parameter $\delta$ allows to freeze the process during a few cycles:

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

The verilog implementation uses a shift register, and only one trigger $t^1$.

```verilog
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 $Op = I_n$, the submodule $Expr$ is called $n$ times from the current module, each time consuming a subset of the inputs and of the outputs.

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

```verilog
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 $\varphi$ modulo $n\delta$ is not null. To get a correct implementation, another trigger and a AND gate are necessary.
Matrix-Matrix Multiplication in Verilog

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.

```verilog
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:

```verilog
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:

```verilog
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.
### 3.2 Specific Rules for MMM

<table>
<thead>
<tr>
<th>Rule</th>
<th>Signification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_i^i \rightarrow I_i$</td>
<td>Remove useless transpositions</td>
</tr>
<tr>
<td>$L_\bar{i}^\bar{i} \rightarrow I_\bar{i}$</td>
<td></td>
</tr>
<tr>
<td>$L_i^j \cdot L_j^i \rightarrow I_{ij}$</td>
<td>Transposition is involutive</td>
</tr>
<tr>
<td>$SER_{1,ij} \rightarrow I_1$</td>
<td>Remove useless serializer</td>
</tr>
<tr>
<td>$DESER_{1,ij} \rightarrow I_1$</td>
<td>Remove useless deserializer</td>
</tr>
<tr>
<td>$ACC_{i,1,j} \rightarrow MEM_{i,j}$</td>
<td>A one time accumulator is a memory</td>
</tr>
<tr>
<td>$MEM_{i,1} \rightarrow I_1$</td>
<td>Remove useless memories</td>
</tr>
<tr>
<td>$SWAP \circ SWAP \rightarrow I$</td>
<td>Swap is involutive</td>
</tr>
<tr>
<td>$S_1 \otimes Expr \rightarrow Expr$</td>
<td>Remove useless kronecker products</td>
</tr>
<tr>
<td>$K_{1,1} \otimes Expr \rightarrow Expr$</td>
<td></td>
</tr>
<tr>
<td>$H_1 \otimes Expr \rightarrow Expr$</td>
<td></td>
</tr>
</tbody>
</table>

| Table 3.2: Additional cleaning rules |

![Diagram of blocking](image)

**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:

$$\mathcal{R}_{m'}^{HH} : MMM_{m,k,n}^{T,\delta} \rightarrow K_{m/m',1} \otimes MMM_{m',k,n}^{T,\delta}$$ (3.1)
Serial Vertical Blocking

The serial vertical blocking is done with the following rule:

\[
\mathcal{R}_{n'}^{V,...}: MMM_{m,k,n}^{t,\delta} \to \begin{cases} 
I_{mn'} \otimes DESER_{n'/n',tn'/n+\delta-1,tn'/n} \circ \\
-MM_{m,k,n'}^{tn'/n,\delta} \circ \\
\left( I_{mk} \otimes MEM_{0,1} \right) \times \left( I_{kn'} \otimes SER_{n'/n',0,tn'/n} \right)
\end{cases}
\]  

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

Transposed Product

So far, we have a serial vertical blocking rule \( \mathcal{R}_{n'}^{V,...} \), 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:

\[
A \cdot B = t ( tB \cdot tA )
\]

This leads to the rule:

\[
\mathcal{R}^T: MMM_{m,k,n}^{T,\delta} \to L_{m,n} \circ MMM_{m,k,m}^{T,\delta} \circ SWAP \circ \left( I_{m,m} \times I_{k,k} \right)
\]

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-wise to form the final matrix.

Parallel

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

\[
\mathcal{R}_{k'}^{D//}: MMM_{m,k,n}^{T,\delta} \to \left( S_{k/k'} \otimes MMM_{m,k',n}^{T,\delta} \right) \circ \left( I_{m,k} \otimes I_{m,k} \right)
\]
3.3 Designs for MMM

Serial

The corresponding serial rule is:

$$\mathcal{R}_{k'}^{D,...} : \text{MMM}^{t,\delta}_{m,k,n} \rightarrow \left\{ \begin{array}{c} \left( \text{I}_{mn} \otimes \text{ACC}_{tk'/k,k'+\delta-1/k'/k,tk'/k} \right) \circ \\
\text{MMM}^{\delta}_{m,k'',n} \circ \\
\left( \left( \text{I}_{nk'} \otimes \text{SER}_{k/k',0,tk'/k} \right) \times \\
L_{n}^{k'} \cdot \left( \text{I}_{nk'} \otimes \text{SER}_{k/k',0,tk'/k} \right) \cdot L_{n}^{k} \right) \end{array} \right. $$

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. $\text{MMM}_{1,k,1}$).

**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-dimensions vectors, we get the formula (after cleaning):

$$\text{MMM}_{1,4,1}^{1,0} \leftarrow S_{4} \otimes \text{MMM}_{1,1,1}^{1,0}$$

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 $o1$:

```verilog
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 synthesiser. Two scenarios may happen:

- The synthesiser 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.

- The synthesiser 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.
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, \( \text{CASC} \) and \( \text{CASCADD} \):

\[
\begin{align*}
\text{CASC}_{n,\delta} &= e_i \odot e_{i+\delta} \odot e_j \odot e_k \\
\text{CASCADD} &= e_i \odot e_j
\end{align*}
\]

The Verilog implementation uses only D-type flip-flops. A \( \text{CASC}_{2,2} \) will be implemented as follows:

```verilog
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)
```

Figure 3.6: Design with cascaded MADDs
3 Matrix-Matrix Multiplication in Verilog

\[
\text{var4} <= \text{var3};
\]
\[
\text{reg [15:0] var5;}
\]
\[
\text{always @(posedge clk)}
\]
\[
\text{var5} <= \text{var4};
\]
\[
\text{reg [15:0] out2;}
\]
\[
\text{always @(posedge clk)}
\]
\[
\text{out2} <= \text{var5};
\]

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

\[
\text{CASCADD}_{n, \delta} = e_0 \odot e_t \odot e_{t-\delta} \odot e_i
\]

The Verilog implementation uses flip-flops and adders. A \(\text{CASCADD}_{2,2}\) is implemented:

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

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:

\[
\mathcal{R}^C : \text{MMM}_{1,1,1}^{1,\delta} \rightarrow \text{CASCADD}_{k,\delta/k} \circ (\text{H}_k \otimes \text{MMM}_{1,1,1}^{1,0}) \circ \left( \text{CASC}_{k,\delta/k} \times \text{CASC}_{k,\delta/k} \right)
\]

This rule builds a scalar product with a gap of 1 cycle, and a variable latency of \(\delta\) cycles.

Pipeline The DSP slices that perform the operations can be internally pipelined. It is possible to indicate to the synthesizer 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:

\[
\mathcal{R}^P : \text{MMM}_{1,1,1}^{1,\delta} \rightarrow D_\delta \circ \text{MMM}_{1,1,1}^{1,0}
\]

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

3.3.2 General MMM

Parallel MMM

A general \(\text{MMM}_{m,k,n}\) can be done by computing \(mn\) scalar products in parallel. This can be done by using \(\mathcal{R}_1^{H,//}\), and \(\mathcal{T} \circ \mathcal{R}_1^{H,//} \circ \mathcal{T}\) 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.
3.3 Designs for MMM

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{R}^C$</td>
<td>$\mathcal{R}^C$</td>
<td>Cascaded adder</td>
</tr>
<tr>
<td>$\mathcal{R}^A$</td>
<td>$\mathcal{R}^P \circ \mathcal{R}_1^{D,//}$</td>
<td>Pipelined adder-tree</td>
</tr>
</tbody>
</table>

Table 3.3: Two families of scalar products

<table>
<thead>
<tr>
<th>Family name</th>
<th>Kernel implemented</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eml</td>
<td>$MMM_{m,k,n}^{x,y,z,p}$</td>
<td>$\mathcal{R}^A \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_1^{T} \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{D,...} \circ \mathcal{R}_x^{V,...} \circ \mathcal{R}_y^{T} \circ \mathcal{R}_z^{V,...}$</td>
</tr>
<tr>
<td>Clr</td>
<td>$MMM_{m,k,n}^{x,y,z,p}$</td>
<td>$\mathcal{R}^C \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_1^{T} \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{D,...} \circ \mathcal{R}_x^{V,...} \circ \mathcal{R}_y^{T} \circ \mathcal{R}_z^{V,...}$</td>
</tr>
<tr>
<td>Apo</td>
<td>$MMM_{m,k,n}^{x,y,z,p}$</td>
<td>$\mathcal{R}^A \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_1^{T} \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{V,...} \circ \mathcal{R}_x^{T} \circ \mathcal{R}_y^{V,...} \circ \mathcal{R}_z^{D,...}$</td>
</tr>
<tr>
<td>AG</td>
<td>$MMM_{m,k,n}^{x,y,z,p}$</td>
<td>$\mathcal{R}^C \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_1^{T} \circ \mathcal{R}_1^{H,//} \circ \mathcal{R}_z^{V,...} \circ \mathcal{R}_x^{T} \circ \mathcal{R}_y^{V,...} \circ \mathcal{R}_z^{D,...}$</td>
</tr>
</tbody>
</table>

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 $\mathcal{R}_x^{V,...}$, $\mathcal{R}_z^{D,...}$ 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 serially block the MMM into a $MMM_{x,y,z,p}$, 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 $N^4$.

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:

```bash
echo 'run -move_first_stage no -move_last_stage ' > mmm.xst
echo 'no -ifn mmm.v -ifmt Verilog ' >> mmm.xst
echo '-oofn 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
trace -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 \( R^A \) and \( R^C \) are used with different level of pipeline.

As expected, the maximum frequency of the adder tree \( (R^A) \) 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 longest path is constant in this case). Then, it starts decreasing, probably because
4 Results

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.

![Performance vs complexity for square MMM](image)

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 designs 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 Results

4.5.1 2x2

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

![MMM 2x2x2 design space](image)

**Figure 4.4: MMM 2x2x2 performances**

![MMM 2x2x2 design space](image)

**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 which matrices are output is divided by 2 or by 4, depending on the serialization 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 $4 \times 4$ matrices.

![MMM 4x4x4 design space](image)

**Figure 4.6: MMM 4x4x4 design space**

![MMM 4x4x4 maximum frequencies](image)

**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
4 Results

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 $8 \times 8$ matrices.

![MMM 8x8x8 design space](image)

**Figure 4.8: MMM 8x8x8 design space**

![MMM 8x8x8 maximum frequencies](image)

**Figure 4.9: MMM 8x8x8 maximum frequencies**

The frequency range is once again lower, between 45 and 240 MHz. Except for highly serialized 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 $16 \times 16$ matrices. Here, designs with low level pipeline ($< 2$) are not displayed.

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 $4 \times 5$ matrix with a $5 \times 7$ matrix.

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.
4.7 Limited Frequency

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 asymmetric 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 completely provided in one cycle. It is possible to handle streaming by using specific operators in the starting formula:

\[ I_2 \otimes SER_{2,1,1} \circ MMM_{2,2,2}^{1,0} \circ \left( I_2 \otimes DESER_{2,0,1} \right) \times \left( I_2 \otimes DESER_{2,0,1} \right) \]

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].

39
Bibliography


