Optimal Streaming Permutations and Transforms: Theory and Implementation

Author(s):
Serre, François

Publication Date:
2019

Permanent Link:

https://doi.org/10.3929/ethz-b-000380039 →

DISS. ETH NO. 25940

OPTIMAL STREAMING PERMUTATIONS AND TRANSFORMS: THEORY AND IMPLEMENTATION

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES OF ETH ZURICH (Dr. Sc. ETH Zurich)

presented by
FRANÇOIS SERRE
Ingénieur diplomé de l'École Polytechnique (X2007)
Master of Science ETH in Computer Science, ETH Zurich
born 6 January 1986
Citizen of France

accepted on the recommendation of
Prof. Dr. Markus Püschel (advisor)
Prof. Dr. Jeremy R. Johnson
Prof. Dr. Paolo Ienne Lopez

April 2019

À la mémoire de mon père.

ABSTRACT

Many algorithms used in hardware applications across disciplines, such as the fast Fourier transform, the Walsh-Hadamard transform or sorting networks share a common structure. They consist of stages of parallel and identical processing elements that each operates on two inputs with different data permutations in between. The symmetry in this structure enables folding to an area-efficient in a streaming architecture that accepts the input over several cycles. However, necessary for folding are streaming permutation circuits that require memory and routing components.

In this dissertation, we focus on the optimal implementation of these algorithms. We provide lower bounds for the implementation of streaming permutations, and propose algorithms that produce solutions that match it (under certain assumptions). We apply these algorithms in the context of a generator capable of implementing the hardware applications previously mentioned. Finally, for a given problem, we address the question of finding an optimal algorithm, i.e., an algorithm which streaming implementations are the cheapest to implement.

RÉSUMÉ

De nombreux algorithmes utilisés dans les circuits intégrés, tels que la transformée de Fourier rapide, la transformée d'Hadamard, ou les réseaux de tri, présentent une structure commune. Ils se composent de plusieurs couches d'éléments de traitement identiques fonctionnant en parallèle, et qui opèrent chacun sur deux entrées. Ces couches sont elles-mêmes séparées par différentes permutations. Les symétries de cette structure permettent de «replier» l'algorithme sur lui-même, permettant ainsi d'obtenir une architecture matérielle économe en ressources et qui travaille sur des données étalées dans le temps. Ceci nécessite de permuter les données au sein de ce flux, ce qui entraîne l'utilisation de mémoire d'une part, et d'éléments capables de diriger spatialement les données d'autre part.

Cette thèse porte sur l'implémentation optimale de tels algorithmes. Nous définissons plusieurs mesures sur les permutations de flux de données, notamment leur *entropie spatiale*, puis montrons qu'elles représentent en fait la quantité de ressources minimale nécessaire à l'implémentation de ces permutations (en terme de nombre de multiplexeurs, ou de quantité de mémoire). Nous proposons ensuite des méthodes qui, sous certaines conditions, permettent de construire des structures optimales. Enfin, pour un problème donné, nous nous interressons à la recherche d'un algorithme optimal, i.e. qui comporte les permutations dont l'implémentation nécessite le minimum de ressources possible.

ACKNOWLEDGMENTS

I owe my advisor Markus Püschel not only for pushing me to pursue this PhD, but as well for his support, his guidance and his patience; often having to proofread the last-minute changes I could make to our publications. I can't imagine what these years would have been without the knowledge he shared, especially in abstract algebra, and the freedom he offered me. I don't think that many advisors would have agreed to let me work on a theorem (Chapter 6) for almost a year, without any guarantee of success. Outside of work, I'm also grateful for the general knowledge and social skills he demonstrated and shared.

I'm particularly lucky for the environment I worked in, and I am thankful to Alen, Daniele, Eliza, Gagandeep, Georg, Joao, Luca, Marcela, Melanie, Tyler and Victoria for the passionate debates we had at lunch, the innumerable evenings we spent together, the sport challenges we met (or not...), the flights, the hikes and the trips we did together. I express as well my fondness to the people I met at ETH and at the ASVZ, to my flatmates at Elsa, to the people of Distran [7], to Cecilia, Erwan, Gemma, Katarina, Michèle, Nicolas, Sophie and Thomas for making these years in Zurich wonderful.

I thank Susanna for her support throughout the end of this PhD, and for assisting me in preparing its defense, though she might not have been overly passionate about the topic.

Je tiens aussi à remercier ceux que je ne vois que trop rarement, mais dont l'amitié dépasse le nombre des kilomètres : les «gros» de la section judo, les guguss, les magnoludoviciens, et les sujets de sa majesté Abdul I^{er}.

En particulier, je remercie Guillaume (un vrai docteur, lui !) et Anne-Sophie pour le soutien et le réconfort qu'ils m'ont apporté au cours de moments douloureux.

Enfin, je remercie ma famille et particulièrement mes parents. Je leur suis infiniment reconnaissant pour leur soutien inconditionnel et les sacrifices qu'ils ont voués à ma réussite. Cette thèse est dédiée à la mémoire de mon père, à qui je dois le goût de la physique et de la science.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In 1822, Joseph Fourier was working on a thermodynamic law that now has his name [55], stating that the heat flux $\vec{\phi}$ in a body is proportional to the gradient in temperature T in this body: $\vec{\phi} = -\lambda \cdot \vec{\text{grad}} T$. Using the conservation of energy ($C \cdot \partial T/\partial t = -\text{div}\vec{\phi}$), this yields the *heat equation* (Chapter 2 of [21]):

$$\frac{\partial T}{\partial t} = \frac{\lambda}{C} \cdot \Delta T.$$

Fourier noticed (Chapter 4 of [21]) that in the case of one spatial dimension[1] x,

$$e^{-\frac{\lambda}{C} \cdot t} \cdot \cos x,\ e^{-3^2 \frac{\lambda}{C} \cdot t} \cdot \cos 3x,\ e^{-5^2 \frac{\lambda}{C} \cdot t} \cdot \cos 5x,\ \ldots$$

were solutions $T(x,t)$ of this equation. To get a general solution for an initial temperature distribution $T(x,0) = T_0$ for $-\pi/2 < x < \pi/2$, Fourier proposed to decompose[2] the latter function into a sum of trigonometric functions $a_n \cos nx$. This allowed him to express a general solution in the form of a sum of the particular solutions he found.

The *discrete Fourier transform* (DFT) is used similarly: it decomposes a discrete signal $x = (x_t)_{0 \leqslant t < N} \in \mathbb{C}^N$ into a sum of trigonometric sequences

$$\left(\frac{y_1}{N} e^{2i\pi t/N}\right)_t,\ \left(\frac{y_2}{N} e^{4i\pi t/N}\right)_t,\ \left(\frac{y_3}{N} e^{6i\pi t/N}\right)_t,\ \ldots.$$

Applying a linear transform to x amounts to applying it to each of these trigonometric sequences. In some cases, this technique can dramatically decrease the computational complexity. For instance, a convolution on one of these sequences can be computed using a single scalar multiplication [43].

The coefficients $y = (y_j)_{0 \leqslant j < N}$ of the sum are given by $y = \text{DFT}_N \cdot x$, where

$$\text{DFT}_N = [\omega^{ij}]_{0 \leqslant i,j < N}, \text{ with } \omega = e^{-2i\pi/N}.$$

As in its origins, the DFT became a "hot topic" when an algorithm to compute it in $O(N \log N)$ was discovered[3] in 1965 [15], the *Fast Fourier transform* (FFT). It is now a ubiquitous tool in signal processing and beyond, used in image and speech processing, radar, wireless communication (e.g., in the LTE standard), and many other domains. Thus, fast and efficient implementations of FFTs, in software and in hardware, and in particular for embedded systems, are of high importance.

The FFT can be implemented using a so-called *butterfly network* as shown in Fig. 1. Note that all dataflows in this document are from right to left because of the corresponding matrix notation introduced later. It consists of stages of parallel and almost identical blocks (the butterflies) that each operates on two inputs with data permutations in between. Many other algorithms used in hardware applications in signal

1 Fourier solves first the problem in stationary conditions in Chapter 3. The problem shown here considers a unique dimensionless spatial coordinate $-\pi/2 \leqslant x \leqslant \pi/2$, with the boundary conditions $T(-\pi/2, t) = T(\pi/2, t) = 0$.

2 He stated that any function was the sum of trigonometric functions, but this turned out to be false. Finding conditions such that a function is the sum of its Fourier serie kept mathematicians like Parseval, Dirichlet or Jordan busy throughout the XIX$^{\text{th}}$ century.

3 Gauß already discovered it in 1805 [26].

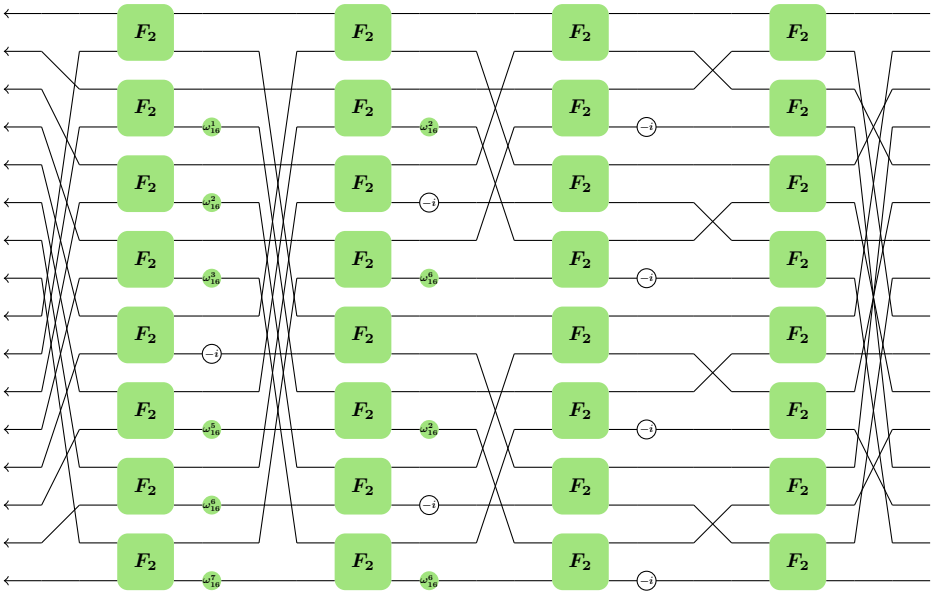


Figure 1: Radix-2 Iterative Cooley-Tukey FFT dataflow (from right to left) operating on $N = 16$ elements.

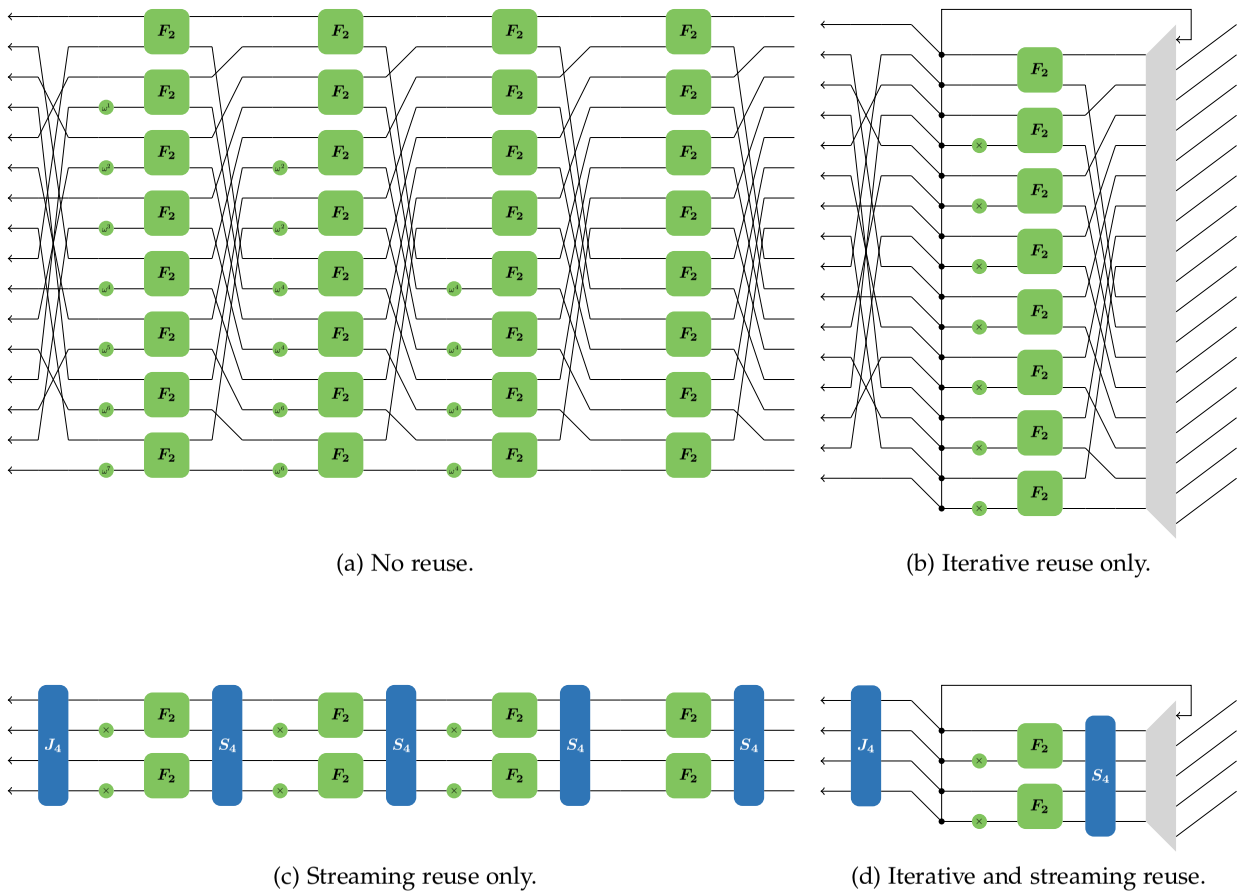
processing, communication, and other domains, share this structure consisting of a network of small processing elements and different intermittent permutations. Examples include the *Walsh-Hadamard transform* (WHT) [29], fast cosine and sine transforms [2], *sorting networks* (SNs) [4, 79], permutation networks [6, 84, 61, 40, 78, 49], and others.

The regular structure offers much flexibility in their hardware implementations and thus there has been extensive work, most focusing on the FFT (e.g., [1, 14, 39, 16, 30, 32, 47, 25, 46, 50]. In particular, [50, 47, 88, 89] propose generators for FFTs and sorting networks that are capable of producing an entire design space of implementations with different trade-offs in performance and resource consumption. These generators are built as a back-end of Spiral, a generator of signal processing libraries tuned for a specific platform [64], and operates with different algorithms represented in a domain specific language (DSL) called SPL. It then exploits different symmetries (or regularities) of these algorithms to obtain a space of relevant designs, as we explain next. The desired design is then output as a *register-transfer level* (RTL) description in the Verilog hardware description language.

Fig. 2a shows the dataflow of a radix-2 Pease FFT [81], a variant of the FFT derived in [60], on $N = 16$ points. The FFT comprises four identical stages (except for the twiddle scaling shown as little circles) of eight parallel butterflies F_2 preceded by a *perfect-shuffle*, followed by the *bit reversal* permutation. This dataflow can be used for a fully-parallel implementation that has high throughput but also high cost.

The cost can be reduced by exploiting the repetitive structure of this FFT. A first method, called *streaming reuse*, "folds the dataflow vertically" to obtain a design like Fig. 2c [57, 50, 47]. Now the circuit operates on streaming data, which means that the dataset arrives on K ports during N/K cycles. In the figure $K = 4$.

Fig. 2a has another symmetry: the first four stages are almost identical. Therefore, it is possible to "fold the dataflow horizontally" to reuse over time a single hardware stage [60] as in Fig. 2b. This is called *iterative reuse*.



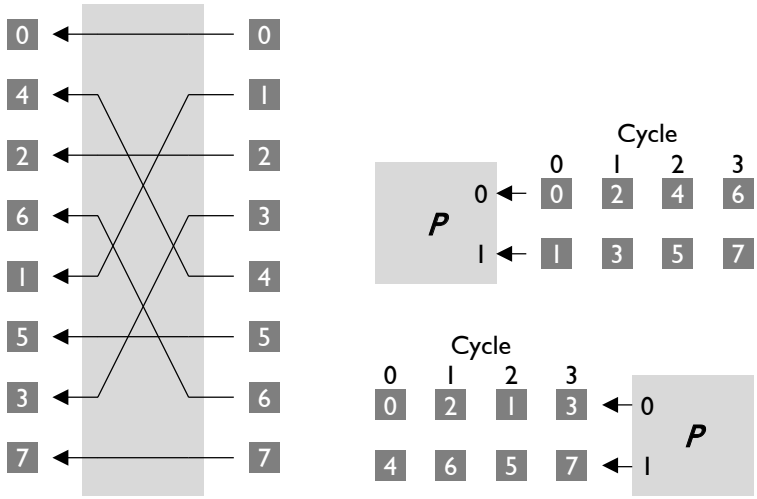
(a) No reuse.

(b) Iterative reuse only.

(c) Streaming reuse only.

(d) Iterative and streaming reuse.

Figure 2: Radix-2 Pease-FFT dataflows operating on $N = 16$ elements with different types of folding [47]. In (a), the design is not folded and consists of 4 stages (each comprising a perfect-shuffle permutation, an array of butterflies F_2 and an element-wise multiplication by constants), followed by a bit-reversal permutation. (b) is horizontally folded: it implements only one instance of this stage that processes the dataset iteratively. (c) is vertically folded: the dataset is input *streamed* in chunks of $2^k = 4$ elements (the *streaming width*) that enter during $N/K = 4$ consecutive cycles. (d) combines both types of folding.



(a) not streaming (b) streaming

Figure 3: Sketch of two implementations of the bit reversal permutation on 2^3 elements. On the left, the structure has as many ports as the dataset. Thus a simple rewiring is enough. On the right side, data are streamed on two ports. Therefore, the dataset enters within 4 cycles (top), and is retrieved within 4 cycles (bottom).

The two types of folding can be combined [50, 47], resulting, for example, in the design shown in Fig. 2d. If fully folded in both dimensions, the design is very compact. In the case shown it contains only two butterflies, two complex multipliers, and the hardware to perform the bit reversal (represented in Fig. 2c and 2d with the blue box labeled with J_4) and the perfect shuffle (labeled with S_4). The work in [50, 47] considers and generates the entire design space given by varying the degree of folding in both dimensions.

1.1 CHALLENGES

1.1.1 *Streaming permutations*

The permutations (or data reorderings) required between the butterfly stages are simple to implement in the case of non-streaming designs (Fig. 2a and 2b). All data are available in one cycle and a hardware implementation consists of a set or wires as shown in Fig. 3a. However, streaming designs (such as Fig. 2c and 2d) require streamed permutation circuits (represented with blue boxes), as data arrive streamed in chunks over several cycles as in Fig. 3b. Their implementation is non-obvious: they require *memory*, as data may now be permuted across cycles, and *routing components* like multiplexers or switches, as elements arriving on a given input port may need to be directed to different output ports. Efficient methods for implementing these have been developed in the literature. There are two classes of implementations. One designs a circuit that can handle any permutation [45, 38], parameterized by the control logic at runtime. However, this flexibility comes at the price of a higher area cost. The second class consists of datapaths that are specialized for the desired permutation [59, 58, 31, 63, 23], which thus reduces cost.

Measuring implementation cost. The cost to implement a streaming permutation can be measured in terms of latency, total memory capacity required, number of independent memory elements, or routing logic complexity. Lower bounds, and imple-

mentation methods that minimize the first three of these measurements are known [38], and it is therefore possible to quantify how hard a given streaming permutation is to implement in this respect, or to estimate how far a given implementation is from optimality. However, quantifying routing complexity remains a challenge. If some methods of implementation pay a particular attention to the number of routing elements used [11, 12, 24, 63], knowing for instance the absolute minimal number of 2-input multiplexers that would be required to implement a given streaming permutation is still an open question.

Best network for streaming. We already presented two different networks for computing a DFT on 16 elements in Fig. 1 and 2a. The latter allows iterative reuse due to its repetitive structure, but it is known that the permutations in the former are cheaper to implement when streamed [46, 47]. A question that arises is, for a given streaming configuration, which butterfly network would be the cheapest to implement. This would require the identification of all possible networks to choose the best.

1.1.2 *Implementation*

The state of the art of the different components needed to implement streaming algorithms keeps improving. We already discussed the streaming permutations, and the existence of different methods to implement them. Arithmetic circuit unit is another important component. For instance, FloPoCo [19] provides an open-source generator for pipelined floating-point arithmetic with arbitrary precision However, no generator to date combines these features with the flexibility offered by [47, 89]. One possible cause is the difficulty of programming a generator capable of mapping a high-level design (as in Figs. 2c and 2d) to a concrete RTL implementation. Some of the challenges are discussed next.

Mismatch of hardware and software datatypes. A first difficulty, common among high-level synthesis (HLS) tools, is the wide diversity of possible datatypes that hardware designs enable. The precision of (unsigned or signed) integers or fixed point numbers can be chosen arbitrarily, in contrast to a small set of choices in software. The same applies to floating-point arithmetic, ranging from IEEE754 to the space covered by FloPoCo [19], which offers arbitrary mantissa and exponent width.

Two different evaluation times. A second issue is that a given function may need to be either evaluated during design generation or implemented in the resulting design, or even partially evaluated during generation and partially implemented.

For instance, the FFT involves multiplications with a set of constants, called *twiddle factors*. A twiddle factor $t_{i,j}$ is a complex number that depends on two parameters: the index i of the element, and the index of the computation stage j. In the case of non-iterative designs (Figs. 2a and 2c), the parameter j is known at generation time, while in iterative scenarios (Figs. 2b and 2d), the design would need to implement a *counter* counting the number of datasets that were already processed by the stage. Similarly, the parameter i is known at generation-time for non-streaming designs (Figs. 2a and 2b) for each different multiplier, while in streaming designs (Figs. 2c and 2d), i depends on the multiplier position, and on a *timer* that counts the number of cycles elapsed since the dataset began to enter. As the computation of a twiddle factor would typically involve a ROM containing different possible values, it is essential to exploit during generation as much as possible the structure of i and j to reduce ROM consumption and DSP slices in case of trivial multiplications.

A typical solution for handling this problem consists of writing and maintaining different versions for each different scenario, which is error-prone and time consuming.

Synchronization issues. Design usually require pipelining to handle the frequency required by the user. Keeping the example of twiddle factors, an inspection of different FFT algorithms shows that most constants are 1, $\mathbf{i}$ (the imaginary unit) or $-\mathbf{i}$, which results in a trivial multiplication that does not require pipelining. However, it is necessary in this case to add supplementary registers if another non-trivial multiplication exists, to keep the whole dataset synchronized.

Additionally, if the twiddle factor computation is done in hardware, it may also require pipelining. As this computation is independent of the input to the FFT, it is possible to initiate it in advance to avoid impacting the global latency of the design. However, this requires a precise cycle tracking to trigger the counter and the timer at the appropriate time.

Handling the latency. As some of the designs produced use a loop (Figs. 2b and 2d), special attention must be paid to guarantee that the latency of the inner structure is long enough to avoid collision between the tail and the head of a given dataset. Additionally, this inner latency determines the minimal time separating two datasets, which must be reported to the user.

1.2 CONTRIBUTIONS

In this thesis, we address the problems above with the following contributions:

1.2.1 *Streaming permutations*

- We introduce a novel metric, the *routing entropy*, that allows to quantify the difficulty of implementing a given streaming permutation in terms of routing elements.
- We present two methods to implement streaming *linear permutations*, a class of permutation that is ubiquitous in signal processing algorithms. The first one yields implementations that have *routing optimality*, which means that no implementation can have less multiplexers. The second method yields *memory optimal* implementations, which means that no implementation can have a lower latency, a lower number of memory elements, or a lower total memory capacity. Additionally, these implementations have the lowest possible number of multiplexers for the particular architecture we consider, and we precisely characterise the cases where routing optimality is reached.
- We propose a method to implement a circuit that can pass several different linear permutations. The circuit has memory optimality, and uses a low number of multiplexers.

1.2.2 *Streaming algorithms*

- We propose a novel small FFT architecture that harnesses the circuit passing several linear permutations to reduce the memory consumption.

- We characterize and propose a method to enumerate all WHT algorithms consisting of stages of columns of butterflies separated by linear permutations.
- We provide a method to search among them those that are the cheapest to implement. Using this method, we generate for small sizes novel butterfly networks that reduce the number of independent memory elements needed, or the number of multiplexers required.

1.2.3 *Hardware generator*

We present a modular hardware generator that generates streaming designs for FFTs, WHTs and sorting networks. It is implemented in Scala [51], and leverages Scala's facilities for embedding DSLs, concepts from lightweight modular staging (LMS) [65] to perform optimization at the DSL levels, and Scala's type system to offer the flexibility discussed above.

1.3 ORGANISATION OF THIS DISSERTATION

This thesis is divided into three parts. The first part considers the theory of streaming algorithms: Chapter 2 discusses the implementation of streaming permutations; Chapter 3 proposes a novel compact streaming FFT architecture, and Chapter 4 considers the search for an optimal WHT streaming algorithm. In the second part (Chapter 5), we present a prototype of a modular generator for streaming hardware. The third part provides two linear algebra tools used in the first part, but that are also of interest on their own: a novel matrix factorization that minimizes the rank of certain blocks (Chapter 6), and a method to identify and enumerate a class of WHT algorithms.

Part I

PERMUTATIONS AT FULL ST(R)EAM

2 STREAMING PERMUTATIONS

A fully parallel hardware implementation of algorithms on large data sets is usually impossible due to the resources it requires. Therefore, the corresponding dataflows need to be *folded* into a streaming architecture, which accepts the input over several cycles. Particularly suited for folding are regular algorithms such as the fast Fourier transform [60] (Fig. 1), Viterbi decoding, or sorting networks [37]. An example of the latter for $N = 8$ elements is shown in Fig. 4a and an associated folded version with streaming width $K = 4$ in Fig. 4b. The folded version halves the number of two input sorters needed for its implementation [89].

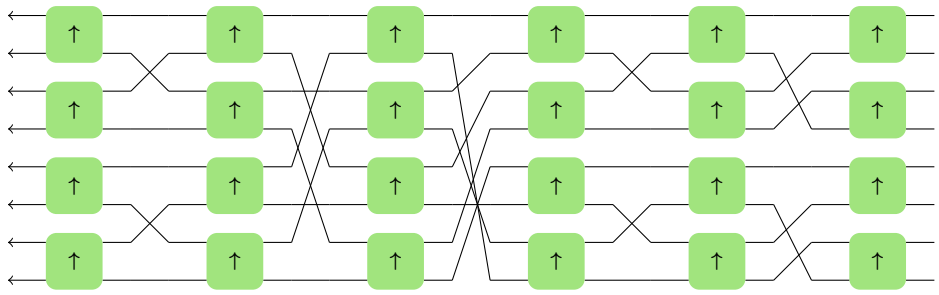
Some permutations are trivial to fold due to their spatial periodicity (e.g., the two rightmost permutations in Fig.4b), i.e., if they perform equal subpermutations on blocks of size K. However, in general, implementing a *streaming permutation* is challenging, as it requires both routing between ports and delays across cycles. We assume only 2-input multiplexers and single-ported RAMs as building blocks, which is well-suited for FPGAs.

In this chapter, that extends the work presented in [68, 71], we quantify the difficulty of implementing a streaming permutation, both in terms of memory and routing logic. We then present a method to implement streamed linear permutations (SLPs) on $N = 2^n$ elements with proven optimality. Linear permutations are the permutations that operate as linear mappings on the bit representation of indices. They include many of the most important occurring permutations including stride permutations and the bit reversal. They are needed in fast Fourier transforms (FFTs; see Fig. 1), fast cosine transforms, sorting networks (see Fig. 4a), Viterbi decoders, and many other applications. Specifically:

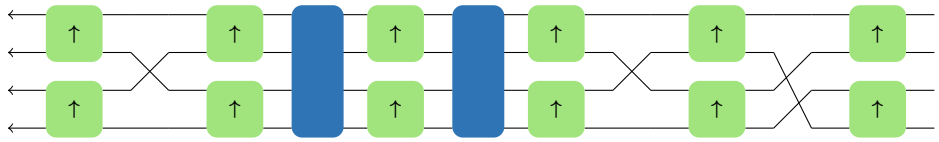
- We prove a lower bound for the routing complexity for a general streaming permutation, i.e., for the number of multiplexers or switches needed.
- We provide a method to derive optimal implementations of SLPs. The method decomposes a given linear permutation into a sequence of spatial and temporal permutations that can be implemented, respectively, as (memoryless) switching networks and banks of RAM. We show that this decomposition is equivalent to a matrix factorization problem in which the minimization of certain ranks of submatrices is equivalent to minimizing the logic of the resulting circuit.
- Finally, we demonstrate our method by generating streamed bit reversal permutations for a Virtex FPGA, and by comparing our optimal solutions to prior art.

2.1 GENERAL STREAMING PERMUTATION

In this section, we give a formal definition of streaming permutations, and provide lower bounds on the logic needed to implement them. The bounds are expressed in terms of two quantities associated with a streaming permutation: its minimal latency



(a) not streaming



(b) streaming ($K = 4$)

Figure 4: A sorting network working on $N = 8$ elements [37]. The blocks with an arrow represent two input sorters. On the top, a fully-parallel implementation. On the bottom, the same implementation "folded" with $K = 4$, allowing to halve the number of sorters [89].

$\delta_{\sigma_N,K}$ and its routing entropy $S_{\sigma_N,K}$. We then define two subclasses of streaming permutations that will be crucial in our approach: temporal and spatial permutations.

2.1.1 *Streaming permutation*

Formally, we define a *streaming permutation* as a pair (σ_N, K), where σ_N is a permutation (or the corresponding permutation matrix[1]) of N words indexed by $0, \ldots, N-1$, and K a positive integer that divides N ($K|N$). K is called the *streaming width*.

An *implementation* of a streaming permutation is a synchronous circuit that has K input and K output ports, and that permutes a dataset of N elements according to σ_N. The dataset enters sequentially in N/K chunks of K elements, and is output similarly (see Fig. 3b). Therefore, the element that enters during the c^{th} input cycle on the p^{th} input port is output during the c'^{th} output cycle on the p'^{th} output port, where

$$c'K + p' = \sigma_N(cK + p).$$

Additionally, we say that an architecture has *full throughput*, if it is capable of handling sequential datasets without interruption.

2.1.2 *Examples*

We provide a few example permutations that we consider:

Identity. The identity matrix of size N is denoted with I_N. An implementation of (I_N, K) directly maps the elements arriving on its K inputs ports to its K output ports.

1 We use a row representation for permutation matrices: if $[p_{i,j}]$ represents σ, then $p_{i,j} = 1$ if $i = \sigma(j)$, and 0 otherwise.

Cyclic shift. We denote with C_N the cyclic shift matrix:

$$C_N = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ 1 & & & \end{pmatrix}. \tag{1}$$

An implementation of (C_N, K) would map the elements of a dataset as follows:

- The element arriving during the input cycle 0 (the first cycle) on the input port 0 (the first port) is output during the last output cycle $(N/K-1)$ on the last output port $(K-1)$.
- Other elements arriving on the first port during the input cycle c are output on the last port at the output cycle $c-1$.
- All other elements, arriving on the port p during the input cycle c are output on port $p-1$ at the output cycle c.

Reversal permutation. The permutation that reverses its inputs is represented by the matrix

$$J_N = \begin{pmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{pmatrix}.$$

An implementation of (J_N, K) maps the element arriving on port p during the input cycle c on output port $K-p-1$ during the cycle $N/K-c-1$.

Stride permutations. If $R|N$, we denote with $L_{N,R}$ the *stride-by-R permutation matrix*, i.e. the matrix representing the transposition of a $(N/R) \times R$-matrix stored in a row-major order:

$$L_{N,R} : i \cdot \frac{N}{R} + j \mapsto j \cdot R + i, \text{ for } 0 \leqslant i < R \text{ and } 0 \leqslant j < N/R. \tag{2}$$

As an example,

$$L_{6,2} = \begin{pmatrix} 1 & . & . & . & . & . \\ . & . & . & 1 & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & 1 & . \\ . & . & 1 & . & . & . \\ . & . & . & . & . & 1 \end{pmatrix}.$$

Additionally, $L_{2R,R}$ is called *perfect-shuffle*.

If $N = K^2$, the streaming permutation $(L_{K^2,K}, K)$ "exchanges space and time": an implementation would map the element arriving on port p during the input cycle c on the output port c during the cycle p.

Half-reversal permutation. If A and B are two matrices, we denote with $A \oplus B$ the *direct sum* of A and B, i.e. the matrix

$$A \oplus B = \begin{pmatrix} A & \\ & B \end{pmatrix}.$$

Additionally, if $(M_i)_{0 \leqslant i < n}$ is a sequence of n matrices, we denote with $\bigoplus_{i=0}^{n-1} M_i$ the matrix

$$\bigoplus_{i=0}^{n-1} M_i = \begin{pmatrix} M_0 & & & \\ & M_1 & & \\ & & \ddots & \\ & & & M_{n-1} \end{pmatrix}.$$

For instance, $I_N \oplus J_N$ is the permutation matrix that represents the permutation that leaves the first half of its inputs unchanged, and that reverses the second half of its inputs. It occurs in fast cosine transforms [77]. Therefore, the streaming permutation $(I_N \oplus J_N, N)$ would alternatively route its input ports directly to its output ports, or reverse them.

2.1.3 *Lower bounds*

We now present four lower bounds that constrain any implementation of a given streaming permutation (σ_N, K). Note that these are derived before we introduce any method of implementation, thus highlighting their architecture agnosticism.

Minimal latency. We denote with $\delta_{\sigma_N,K}$ the maximal difference an element can have between its input cycle c and its output cycle c':

$$\delta_{\sigma_N,K} = \max c - c' = \max_{0 \leqslant i < N} \lfloor i/K \rfloor - \lfloor \sigma_N(i)/K \rfloor.$$

Proposition 1. *The latency of an implementation is bounded by* $\delta_{\sigma_N,K}$.

Proof. A shorter latency would mean that an element would be output before entering the circuit. □

As an example, we consider the permutation matrix

$$C_6 = \begin{pmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ 1 & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

and the streaming permutation $(C_6, 3)$. It satisfies, for $0 \leqslant i < N$

$$\lfloor i/K \rfloor - \lfloor \sigma_N(i)/K \rfloor = \begin{cases} -1, & \text{if } i = 0, \\ 1, & \text{if } i = 3, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, we have $\delta_{C_6,3} = 1$, meaning that any implementation of this streaming permutation has at least a latency of 1 cycle.

Minimal memory capacity. The minimal memory capacity that an implementation must have is directly linked to the minimal latency:

Proposition 2. *An implementation requires a memory capacity of at least* $K \cdot \delta_{\sigma_{N,K}}$ *words.*

Proof. By definition, $\delta_{\sigma_{N,K}} \leqslant N/K$. This means that during the first $\delta_{\sigma_{N,K}}$ input cycles, $K \cdot \delta_{\sigma_{N,K}}$ elements enter the circuit. As the output begins at least at the $\delta^{\text{th}}_{\sigma_{N,K}}$ cycle, the circuit has to store them. □

Our example $(C_6, 3)$ therefore requires a memory capacity of at least 3 words for any of its implementation.

Minimal number of RAM banks. Due to the required throughput, when an implementation requires memory, this memory must be distributed into a minimal number of banks:

Proposition 3. *If* $\delta_{\sigma_{N,K}} > 0$*, the memory used in an implementation needs to be distributed across a minimum of* K *different banks.*

Proof. As $\delta_{\sigma_{N,K}} > 0$, all the K elements arriving during the first input cycle need to be stored in memory. As a memory bank supports only one input element per cycle, K of these are required. □

Back to our example $(C_6, 3)$, this proposition shows that any of its implementation requires at least 3 independent memory banks.

These three first bounds are sharp, as the implementations proposed in [38] match them (if the registers used for pipelining are not considered). We call an implementation *memory optimal* if these three bounds are matched.

Minimal number of multiplexers. For a given streaming permutation (M_N, K), we consider the $K \times K$-matrix

$$R_{M_N,K} = [r_{p',p}],$$

where $r_{p',p}$ the number of elements of a dataset that arrive on the p^{th} input port and that leaves on the p'^{th} output port. Formally,

$$r_{p',p} = |\{(c, c') \mid c'K + p' = \sigma_N(cK + p)\}|.$$

It can be obtained by blocking M_N into $K \times K$ blocks, and summing them all together. This matrix is a semi-magic square [45][2], as a total of N/K elements of a dataset transit through each input port p, and through each output port p. Dividing it by the magic constant yields a bistochastic matrix:

$$\Omega_{M_N,K} = [\omega_{p',p}] = \frac{K}{N} \cdot R_{M_N,K}. \tag{3}$$

We call this matrix the *routing matrix* of the streaming permutation.

In our example $(C_6, 3)$, we have

$$C_6 = \left(\begin{array}{ccc|ccc} \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ 1 & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right),$$

2 $R_{M_N,K}$ is denoted with $\pi_K(M_N)$ in [45].

and thus

$$R_{C_6,3} = \begin{pmatrix} \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & 2 & \cdot \\ \cdot & \cdot & 2 \\ 2 & \cdot & \cdot \end{pmatrix}$$
$$= 2 \cdot C_3.$$

The routing matrix of $(C_6, 3)$ is therefore $\Omega_{C_6,3} = 1/2 \cdot R_{C_6,3} = C_3$.

Note that $\omega_{p',p}$ is the probability that a random element (uniformly chosen) arriving on input port p (resp. coming out of output port p') has to be output on port p' (resp. originates from input port p).

We are interested in measuring "how much variety" in mapping between ports is required across the different cycles. Therefore, it is natural to introduce the *spatial entropy* of the streaming permutation as[3]

$$S_{\sigma_N,K} = -\sum_{0 \leqslant p,p' < K} \omega_{p',p} \log_2 \omega_{p',p}. \tag{4}$$

Interestingly enough, this spatial entropy turns out to be a lower bound on the number of 2-input multiplexers that an implementation requires. The following theorem[4] is an important contribution of this thesis; to the author's knowledge, no such bound currently exists in the literature.

Theorem 1. *A full-throughput implementation of a streaming permutation* (σ_N, K) *that uses only 2-input multiplexers for routing has at least* $\lceil S_{\sigma_N,K} \rceil$ *multiplexers.*

Proof. Given such an implementation, we first enumerate, for a given output port p', the number of multiplexers $\ell_{p'}$ that the elements that arrive on this port had to go through all together. We denote with $P_{p'}$ the set of input ports these elements are coming from.

As only 2-input multiplexers are used for routing, we can represent the different paths the elements can take in the implementation using a direct acyclic graph (DAG) where

- output ports are sinks,
- multiplexers are nodes, each of them having two outgoing edges named 0 and 1 that point to the multiplexer or input port that are connected to its two inputs, and
- input ports are sources with one outgoing edge pointing at the multiplexer or input port they are connected to.

Other elements (memory banks, buffers, ...) do not appear in this graph.

In this DAG, each input port in $P_{p'}$ is reachable from the output port p'. For each of them, we pick one of the shortest path from the output port p', and encode it as a sequence of bits, where each bit describes which multiplexer input was taken, from the output port p' to the input port p. This defines a prefix code for the words in $P_{p'}$,

3 We implicitly extend $x \mapsto x \log_2 x$ with $0 \mapsto 0$.

4 Part of the proof is built on top of some unpublished ideas of Thomas Holenstein.

Streaming permutation (M_N, K)	Minimal latency $\delta_{M_N,K}$	Routing matrix $\Omega_{M_N,K}$	Routing entropy $S_{M_N,K}$
(C_N, K)	1	C_K	0
(J_N, K)	$N/K-1$	J_K	0
$(I_N \oplus J_N, N)$	0	$\frac{1}{2}(I_N + J_N)$	$2 \cdot \lceil N/2 \rceil$

Table 1: Bounds and routing matrix for some streaming permutations.

where the length $\ell_{p,p'}$ of each code word is the minimal number of multiplexers from the corresponding input port to the output port p'. We have therefore:

$$\ell_{p'} \geqslant \sum_{p \in P_{p'}} r_{p',p} \ell_{p,p'} = \frac{N}{K} \sum_{p \in P_{p'}} \omega_{p',p} \ell_{p,p'}.$$

This last sum can be seen as the expected word length of the code, where $\omega_{p',p}$ would be the probability of appearance of p. It is lower-bounded by the corresponding entropy [76]:

$$\ell_{p'} \geqslant -\frac{N}{K} \sum_{p \in P_{p'}} \omega_{p',p} \log_2 \omega_{p',p} = -\frac{N}{K} \sum_{0 \leqslant p < K} \omega_{p',p} \log_2 \omega_{p',p}.$$

A multiplexer outputs at most one element per cycle. Thus, the multiplexers of the circuit all together spend at least $\ell_{p'}$ cycles per dataset to route the elements that will eventually arrive on port p', and therefore at least

$$\sum_{0 \leqslant p' < K} \ell_{p'} \geqslant -\frac{N}{K} \sum_{0 \leqslant p,p' < K} \omega_{p',p} \log_2 \omega_{p',p} = \frac{N}{K} \cdot S_{\sigma_N,K}$$

cycles to route the complete dataset.

As the implementation has full-throughput, a complete dataset needs to be permuted every N/K cycles. A minimum of $S_{\sigma_N,K}$ multiplexers are therefore required. □

Corollary 1. *A full-throughput implementation of a streaming permutation* (σ_N, K) *that uses only* 2×2*-switches for routing has at least* $\lceil S_{\sigma_N,K}/2 \rceil$ *of them.*

Proof. A 2×2-switch can be implemented using two 2-input multiplexers. □

As we will see later, this last bound is sharp for the subclass of streaming linear permutations, and in all the examples given. This may not hold in the general case[5]. We call an implementation that matches this bound *routing optimal*.

In our example, $(C_6, 3)$ has a spatial entropy of 0 bit. This means that an implementation doesn't require any routing element. Other examples are shown in Table 1.

5 For example, we have lost some precision in the proof by removing ceiling operators for readability reasons.

2.1.4 *Spatial and temporal permutations*

The bounds in Section 2.1.3 rely on two quantities, $\delta_{\sigma_N,K}$ and $S_{\sigma_N,K}$. In this section, we identify streaming permutations for which the value of these numbers become 0. This classification shares similarities[6] with the one introduced in [57].

Spatial permutations. The first category of streaming permutation that we consider are those that permute their elements within cycles:

Proposition 4. *Given a streaming permutation* (M_N, K)*, the following propositions are equivalent:*

1. (M_N, K) *can be implemented without memory.*
2. $\delta_{M_N,K} = 0$.
3. *An implementation of* (M_N, K) *only permutes elements within cycles.*
4. M_N *is* $K \times K$*-block diagonal: there exist* N/K $K \times K$ *permutation matrices* $Q_0, \dots, Q_{N/K-1}$ *such that*

$$M_N = \bigoplus_{i=0}^{N/K-1} Q_i. \tag{5}$$

We call such a streaming permutation a *spatial permutation*.

Proof. 1 $\implies$ 2 is the contraposition of Proposition 2.

2 $\implies$ 3: We first compute the sum of the differences between all the input cycles and the output cycles for an implementation (σ_N, K):

$$\begin{aligned}
\sum_{0 \leqslant i < N} c - c' &= \sum_{0 \leqslant i < N} (\lfloor i/K \rfloor - \lfloor \sigma(i)/K \rfloor) \\
&= \sum_{0 \leqslant i < N} \lfloor i/K \rfloor - \sum_{0 \leqslant i < N} \lfloor \sigma(i)/K \rfloor \\
&= \sum_{0 \leqslant i < N} \lfloor i/K \rfloor - \sum_{0 \leqslant i < N} \lfloor i/K \rfloor \\
&= 0.
\end{aligned}$$

Assuming that $\delta_{\sigma_N,K} = 0$ means by definition that $c - c' \leqslant 0$. As a sum of negative numbers is null only if all terms are null, this means that for all the elements, $c - c' = 0$, i.e. that (σ_N, K) does not permute elements across different cycles.

3 $\implies$ 4 is trivial.

4 $\implies$ 1: A streaming permutation (M_N, K) such that $M_N = \bigoplus_{i=0}^{N/K-1} Q_i$, where Q_i is a $K \times K$ matrix can be implemented using a (memoryless) $K \times K$-permutation network, such as the one described in [84]. It suffices to configure it to perform the permutation Q_c on its inputs during cycle c. □

We denote with $A \otimes B$ the *Kronecker product* of A and B, i.e. the matrix

$$A \otimes B = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,q}B \\ \vdots & \ddots & \vdots \\ a_{p,1}B & \cdots & a_{p,q}B \end{pmatrix}, \text{ where } A = [a_{i,j}]_{1 \leqslant i \leqslant p, 1 \leqslant j \leqslant q}.$$

6 We allow temporal permutations to perform a steady spatial permutation.

If a spatial permutation (M_N, K) satisfies

$$M_N = \bigoplus_{i=0}^{N/K-1} Q = I_{N/K} \otimes Q, \tag{6}$$

where Q is a $K \times K$ matrix, then we call this streaming permutation a *steady spatial permutation*. In this case, an implementation would perform the same permutation on every chunk of a dataset at every cycle. It can therefore be implemented by connecting directly the input ports to the output ports.

As shown in Table 1, $(I_N \oplus J_N, N)$ is an example of temporal permutation. $I_N \oplus J_N$ has already the form (5). It can be optimally implemented using $2 \cdot \lfloor N/2 \rfloor$ 2-input multiplexers as follows. If N is odd, then the output port $\lfloor N/2 \rfloor$ is directly connected to the input port $\lfloor N/2 \rfloor$. Other output ports p' are connected to a multiplexer that routes elements from the input port p' during the first cycle, and from the input port $N - p' - 1$ during the second cycle.

Temporal permutations. Conversely, we consider streaming permutations that always route elements coming from a given input port to the same output port:

Proposition 5. *Given a streaming permutation* (M_N, K)*, the following propositions are equivalent:*

1. (M_N, K) *can be implemented without any routing element (multiplexer, switches).*
2. $S_{M_N,K} = 0$.
3. $\Omega_{M_N,K}$ (3) *is a permutation matrix.*
4. *There exist* K $N/K \times N/K$ *permutation matrices* $Q_1, \dots, Q_K$ *such that*

$$M_N = I_{N/K} \otimes \Omega_{M_N,K} \cdot L_{N,K} \cdot \left(\bigoplus_{i=1}^{K} Q_i \right) \cdot L_{N,N/K}. \tag{7}$$

We call such a streaming permutation a *temporal permutation*.

Proof. 1 $\implies$ 2 is the contraposition of Theorem 1.

2 $\implies$ 3: $x \mapsto x \log_2 x$ is negative for $0 \leqslant x \leqslant 1$, and is null only for $x = 0$ and $x = 1$. If $S_{M_N,K} = 0$, then we have

$$\sum_{0 \leqslant p,p' < K} \omega_{p',p} \log_2 \omega_{p',p} = 0.$$

As a sum of negative numbers, it is null if and only if all summands are null, i.e., $\omega_{p',p} \in \{0, 1\}$. As $\Omega_{M_N,K}$ is by definition bistochastic, it is a permutation matrix.

3 $\implies$ 4: If (M_N, K) is such that $\Omega_{M_N,K}$ is a permutation matrix, then it means that all the elements arriving on port p are routed to the port p', where $\omega_{p',p} = 1$. Therefore, the streaming permutation

$$\left((I_{N/K} \otimes \Omega_{M_N,K})^{-1} \cdot M_N, K \right) = \left(I_{N/K} \otimes \Omega_{M_N,K}^T \cdot M_N, K \right)$$

keeps elements on the same port. A computation then shows that

$$L_{N,N/K} \cdot \left(I_{N/K} \otimes \Omega_{M_N,K}^T \right) \cdot M_N \cdot L_{N,K}$$

is $(N/K) \times (N/K)$-block diagonal.

$4 \implies 1$: We consider a streaming permutation (M_N, K), where

$$M_N = I_{N/K} \otimes \Omega_{M_N,K} \cdot L_{N,K} \cdot \left(\bigoplus_{i=1}^{K} Q_i \right) \cdot L_{N,N/K},$$

and will implement it without using any routing element.

In the case where $\delta = 0$ (or equivalently, if $Q_i = I_{N/K}$ for all i), then (M_N, K) is a steady spatial permutation and can be implemented as before. Otherwise, we will implement it using K RAM banks, each capable of storing δ elements.

Each input port p is directly connected to the write port of the p^{th} bank, and the read port of this bank is connected to the output port that corresponds to the image of p through the permutation associated with $\Omega_{M_N,K}$.

Incoming elements are first written linearly in the bank. After δ cycles, the output starts, and elements are retrieved in the permuted order, i.e. at the address that corresponds to the image of the output cycle through the permutation associated with Q_p^{-1}. As we are using single ported RAM banks, incoming elements are written at the address where a previous element is being read, and the read address of this new element must be computed accordingly. For example, if $\delta = K$, the first set is written linearly in the memory, then the second set is written where the first set is read, i.e. at address corresponding to the image of the cycle through the permutation associated with Q_p^{-1}. The i^{th} set is then read at the address corresponding to the permutation Q_p^{-i}. This address can be pre-computed for every banks, and stored in a ROM. □

Streaming cyclic shift (C_N, K) and streaming reversal (J_N, K) are examples of temporal permutations.

A possible decomposition (7) for (C_N, K) is given by

$$C_N = I_{N/K} \otimes C_K \cdot L_{N,K} \cdot (C_{N/K} \oplus I_{N-N/K}) \cdot L_{N,N/K}.$$

It can be optimally implemented using the method described in Proposition 5, with a latency of 1 cycle. This implementation would consist of K memories of 1 word, each having their write port connected to one input port. The first one would store the element arriving during the first input cycle, and would output directly (without delay) all the other elements. The first element would then be output during the last output cycle (corresponding to the first input cycle for the next dataset). The other memories would simply delay all elements by one cycle. The read ports of these memories would then be permuted using a hard-wired cyclic shift C_K, and connected to the output ports.

For the streaming reversal (J_N, K), a possible decomposition (7) is

$$J_N = I_{N/K} \otimes J_K \cdot L_{N,K} \cdot (I_K \otimes J_{N/K}) \cdot L_{N,N/K}.$$

The method described in Proposition 5, yields an optimal implementation with a latency of $N/K - 1$ cycles. This implementation would consist of K memories of $N/K - 1$ words, each having their write port connected to one input port. These memories would write all the first $N/K - 1$ elements they receive, pass-through the last one, and would then read elements in the reversed order. The read ports of these memories would then be permuted using a hard-wired reversal J_K, and connected to the output ports.

2.2 STREAMING LINEAR PERMUTATION

As mentioned in the introduction, we focus on the subclass of *streaming linear permutations* (SLPs). These are characterized by their permutation being linear, which we define next. Then, we will see how the concepts of spatial and temporal permutations translate to SLPs, before proposing methods to implement them optimally. Our work builds on and extends [57, 63].

The size of linear permutations is always a power of two, $N = 2^n$, and, as $K|N$, we have $K = 2^k$. For convenience, we also write $2^t = N/K = 2^{n-k}$ for the number of cycles needed to input the dataset. Therefore, we have $n = k + t$.

2.2.1 *Linear permutations*

We introduce the class of *linear permutations* using first some important examples.

Bit-reversal permutation. Used in FFTs, the bit-reversal permutation has been studied extensively [36]. It maps each element to the position given by reversing the binary representation of its index. Formally, we denote the binary representation of an index i with a column vector i_b of n bits[7], such that the most significant bit is at the top. For example, if $n = 3$,

$$6_b = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix},$$

which the bit reversal maps by flipping upside down to obtain 3_b. Formally, it maps positions as $i_b \mapsto i'_b = J_n \cdot i_b$. Here, the bit matrix J_n describes how the bit reversal operates "on the bits," and should not be confused with the $2^n \times 2^n$ permutation matrix that encodes how the data is mapped.

In the Pease FFT of a general radix 2^r, $r|n$, the bit reversal operates at coarser granularity and is given by $J_{n,r} = J_{n/r} \otimes I_r$.

Perfect shuffle. The perfect shuffle $L_{2^n,2^{n-1}}$ (2) is the permutation that interleaves the first and second half of a list of 2^n elements. It appears in the first four stages of Fig. 2a. For instance, if we consider 8 elements indexed from 0 to 7, these get rearranged such that the element i is mapped to the position $2i$ if $i < 4$, or $2i - 7$ otherwise. If we write the binary representation i_b, we get

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix},$$

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \text{ and } \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

We observe that the perfect shuffle cyclically rotates the binary representation i_b of its indexes.

7 $i_b \in \mathbb{F}_2^n$, where $\mathbb{F}_2$ is the Galois field with two elements, 0 and 1, considered as the two states of a bit. Multiplying and adding bits respectively amounts to ANDing them and XORing them.

More generally, for a set of 2^n elements, the perfect shuffle maps an index $0 \leqslant i < 2^n$ to the index j such that

$$j_b = C_n \cdot i_b.$$

In summary, the invertible $n \times n$ bit matrix C_n defines the perfect shuffle permutation on 2^n elements, which we denote with $L_{2^n,2^{n-1}} = \pi(C_n)$.

Linear permutations. In general, a *linear permutation* [61, 40, 63] π on 2^n elements is a permutation such that there exists an $n \times n$ invertible bit matrix[8] P that satisfies, for $0 \leqslant i < 2^n$,

$$\pi : i \mapsto j \Leftrightarrow j_b = P \cdot i_b. \tag{8}$$

Conversely, for any $n \times n$ invertible bit-matrix P, there is a unique linear permutation that satisfies (8), and we denote it with $\pi(P)$.

For a given n, there are a total of $\prod_{i=0}^{n-1}(2^n - 2^i)$ such P, and thus linear permutations. This means most permutations on 2^n points are not linear (e.g., linear requires that zero is mapped to zero. Therefore, when $n > 0$, the permutation matrices J_{2^n} and C_{2^n} represent non-linear permutations.). But, interestingly, many permutations in signal processing algorithms are linear. Examples include permutations appearing in FFTs, fast cosine transforms, Viterbi decoders, sorting networks, filter banks, and many others.

For instance, if

$$V_n = \begin{pmatrix} 1 & & \\ \vdots & \ddots & \\ 1 & & 1 \end{pmatrix},$$

then $\pi(V_3)$ is the permutation: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 2$, $3 \mapsto 3$, $4 \mapsto 7 \mapsto 4$, $5 \mapsto 6 \mapsto 5$. More generally, $\pi(V_n)$ is the half-reversal permutation we saw earlier:

$$\pi(V_n) = I_{2^{n-1}} \oplus J_{2^{n-1}}.$$

Properties of linear permutations. Composing two linear permutations corresponds to multiplying the associated matrices:

$$\pi(P) \circ \pi(Q) = \pi(PQ).$$

Additionally, we have $\pi(I_n) = I_{2^n}$ and therefore[9]:

$$\pi(P^{-1}) = \pi(P)^{-1}.$$

As an example, every stride permutation on 2^n elements $L_{2^n,2^{n-r}}$ is a power r of the perfect-shuffle $\pi(C_n)$. Therefore, these are linear permutations as well with the associated matrix C_n^r. These are the permutations that appears between stages of a radix 2^r Pease FFT.

Additionally, π satisfies

$$\pi(P \oplus Q) = \pi(P) \otimes \pi(Q).$$

8 Mathematically, $P \in GL_n(\mathbb{F}_2)$.

9 π is a group-homomorphism.

As a consequence, a steady spatial SLP (6) can be written as $(\pi(I_t \oplus Q), 2^k)$, where Q is a $k \times k$ invertible bit-matrix.

Bit-permutations. If the invertible bit-matrix P is itself a permutation, $\pi(P)$ is called a *bit-permutation*. Bit-reversals ($\pi(J_{n,r})$) and stride permutations ($\pi(C^r_n) = L_{2^n,2^{n-r}}$) are examples of these. The "half-reversal" ($\pi(V_n) = I_{2^{n-1}} \oplus J_{2^{n-1}}$) is an example of linear permutation that is not a bit-permutation. These are also known as PIPID in [40] or BP class in [49].

Bit-permutations are closed under multiplication[10], and there are $n!$ of them.

Bit-permutations $<$ linear-permutations $<$ permutations.

2.2.2 *Routing entropy for SLPs*

If $N = 2^n$ data are streamed through $K = 2^k$ ports over $N/K = 2^t$ cycles, $n = t + k$, then the cycle during which an element arrives corresponds to the t most significant bits of its index, while the port corresponds to the k least significant bits. For instance, for $t = k = 2$, the element indexed with

$$11_b = \begin{pmatrix} 1 \\ 0 \\ \hline 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2_b \\ \hline 3_b \end{pmatrix}$$

arrives during the second cycle on the third port. This suggests blocking the matrix P of an SLP $(\pi(P), 2^k)$ to be implemented as

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix}, \text{ such that } P_1 \text{ is } k \times k. \tag{9}$$

Namely, an element arriving in cycle c on port p is output at port p' during the cycle c', where

$$p'_b = P_1 p_b + P_2 c_b \text{ and} \tag{10}$$
$$c'_b = P_4 c_b + P_3 p_b. \tag{11}$$

Routing entropy. The block P_2 is directly linked with the routing entropy of the SLP:

Proposition 6. *The routing entropy of an SLP* $(\pi(P), 2^k)$ *is*

$$S_{\pi(P),2^k} = 2^k \cdot \operatorname{rank} P_2.$$

Proof. If we accumulate across cycles for all inputs at port p, the bit representations of the corresponding output ports, we get, using (10),

$$\{P_1 p_b + P_2 c_b \mid 0 \leqslant c < 2^t\} = P_1 p_b + \operatorname{im} P_2.$$

This set (as a coset of direction $\operatorname{im} P_2$) contains $2^{\operatorname{rank} P_2}$ elements. This means that each input port has to communicate with $2^{\operatorname{rank} P_2}$ different output ports. In other words, each column of $\Omega_{\pi(P),2^k}$ contains $2^{\operatorname{rank} P_2}$ non-zero elements.

10 These are a subgroup of linear permutations.

Let now p' be one of the $2^{\operatorname{rank} P_2}$ possible output ports for an element from input port p. Further, let $\bar{c}$ be an input cycle of an arbitrary element which transits from p to p'. The set of cycles for which an element transits from p to p' is:

$$\{c_b \mid p'_b = P_1 p_b + P_2 c_b\} = \bar{c}_b + \ker P_2.$$

This set (as a coset of direction $\ker P_2$) contains $2^{t-\operatorname{rank} P_2}$ elements. This means that each non-zero element of $\Omega_{\pi(P),2^k}$ has the value

$$\omega_{p',p} = \frac{2^{t-\operatorname{rank} P_2}}{2^t} = 2^{-\operatorname{rank} P_2}.$$

Using the definition of the routing entropy (4), we get:

$$\begin{aligned} S_{\sigma_N,K} &= -\sum_{0 \leqslant p,p' < K} \omega_{p',p} \log_2 \omega_{p',p} \\ &= -2^k \cdot 2^{\operatorname{rank} P_2} \cdot 2^{-\operatorname{rank} P_2} \cdot \log_2(2^{-\operatorname{rank} P_2}) \\ &= 2^k \operatorname{rank} P_2. \end{aligned}$$

□

This directly yields routing bounds for SLPs, using Theorem. 1:

Corollary 2. *A full-throughput implementation of an SLP $(\pi(P), 2^k)$ requires at least:*

- $2^k \operatorname{rank} P_2$ *2-input multiplexers, or*
- $2^{k-1} \operatorname{rank} P_2$ *2×2-switches.*

2.2.3 *Spatial and temporal SLPs*

In this section we characterize the SLPs that are spatial or temporal as defined in Section 2.1.4. This can easily be done using the block structure of P in (9). Further, we show how a spatial SLP can be optimally implemented, i.e., using the minimal number of 2×2-switches.

Spatial SLPs. We already encountered the case of steady spatial SLPs, which were characterized by the bit-matrix

$$P = I_t \oplus P_1 = \begin{pmatrix} I_t & \\ & P_1 \end{pmatrix}.$$

Proposition 7. *Spatial SLPs are the SLPs $(\pi(P), 2^k)$ that satisfy*

$$P = \begin{pmatrix} I_t & \\ P_2 & P_1 \end{pmatrix}. \tag{12}$$

Proof. Spatial permutations are the streaming permutations that permute only within cycles. Therefore, P must leave the upper t bits c_b of each address unchanged in (11), i.e., satisfy $P_4 c_b + P_3 p_b = c_b$, which yields the expected form. □

We show how to optimally implement a given spatial permutation using a switching network (SNW) with $\operatorname{rank} P_2 \cdot 2^{k-1}$ 2×2-switches, thus matching the lower bound of Corollary 2. The network we construct is an Omega network [40] with $k - \operatorname{rank} P_2$ stages removed. An optimal solution is already given in [63]; our description here is somewhat simpler and included for completeness:

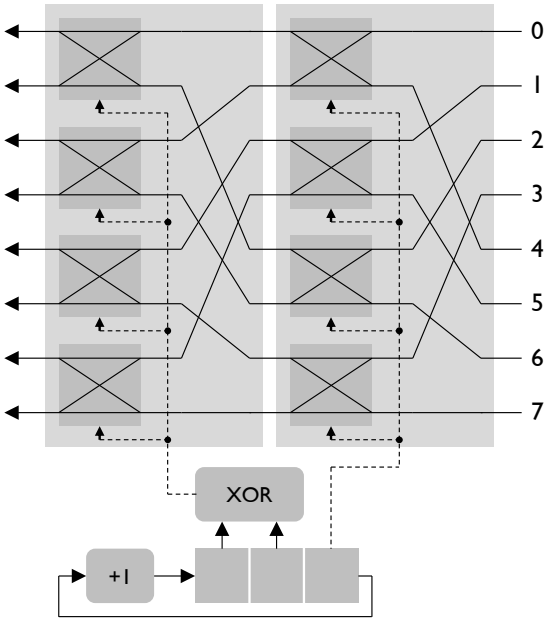


Figure 5: An SNW consisting of two Omega network stages. Each stage contains a perfect shuffle followed by a column of 2^{k-1} switches controlled by a single common bit. Here, the first stage is controlled by a single bit of a counter, while the second one is controlled by the sum of the two other bits of this counter.

Proposition 8. *A spatial SLP* $(\pi(P), 2^k)$ *can be implemented optimally, i.e. using* $2^{k-1} \cdot \operatorname{rank} P_2$ 2×2*-switches.*

Note that this number matches the lower bound in Corollary (2).

Proof. A stage of an Omega network consists of a perfect shuffle followed by a column of 2^{k-1} 2×2-switches: see Fig. 5, which shows 2 stages. We first consider one column of switches. If these switches are all controlled by a common bit, then, when this bit is set, pairs of elements are exchanged:

$$\begin{cases} p \mapsto p+1 & \text{if } p \text{ is even} \\ p \mapsto p-1 & \text{if } p \text{ is odd,} \end{cases} \tag{13}$$

otherwise the column of switches leaves the data unchanged.

We add a counter c of t bits that is incremented at every cycle. Then, for a fixed vector v of t bits, it is possible to compute $c_b \cdot v$ using xor gates, and we use the result to control the column of switches. This structure performs the permutation (13) when $c_b \cdot v = 1$, and does nothing otherwise. In other words, we have implemented $\pi(K_v)$, where

$$K_v = \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v^T & & & 1 \end{pmatrix}.$$

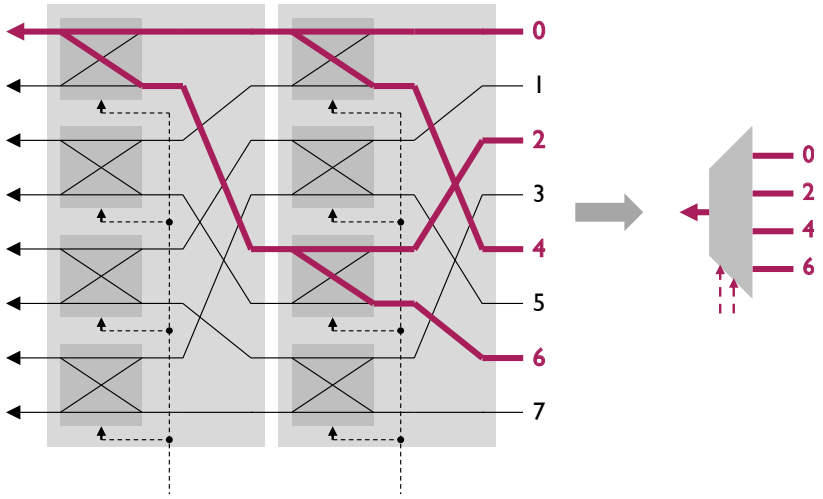


Figure 6: Implementation of the first output port of a switching network using a 4-to-1 multiplexer.

The perfect shuffle that precedes within the stage is a steady spatial permutation, i.e., a rewiring. Therefore, with our formalism, one stage in Fig. 5 is described by the matrix:

$$S_\nu = K_\nu \cdot \begin{pmatrix} I_t & \\ & C_k \end{pmatrix}.$$

We now construct an implementation for a spatial permutation given by (12). First, we find an invertible $k \times k$-matrix L such that LP_2 has $\operatorname{rank} P_2$ non-zero lines ν_i^T at the top (Gauss elimination):

$$LP_2 = \begin{pmatrix} \nu_1^T \\ \vdots \\ \nu_{\operatorname{rank} P_2}^T \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Direct computation shows that:

$$P = \begin{pmatrix} I_t & \\ & L^{-1} C_k^{k-\operatorname{rank} P_2} \end{pmatrix} S_{\nu_{\operatorname{rank} P_2}} \dots S_{\nu_1} \begin{pmatrix} I_t & \\ & LP_1 \end{pmatrix}.$$

This yields an implementation with $\operatorname{rank} P_2$ Omega network stages framed by two rewirings. Thus, the number of switches used is $\operatorname{rank} P_2 \cdot 2^{k-1}$. □

Finally, 2×2-switches can easily be implemented using two 2-to-1 multiplexers. However, some platforms may support larger multiplexers more efficiently. In this case, it is possible to group several switches of different stages as shown in Fig. 6 with an example.

Temporal SLP. Proposition 6 directly yields the structure of temporal SLPs:

Corollary 3. *Temporal SLPs are the SLPs* $(\pi(P), 2^k)$ *that satisfy*

$$P = \begin{pmatrix} P_4 & P_3 \\ & P_1 \end{pmatrix}. \tag{14}$$

The implementation of a temporal SLP is not fundamentally different from a general temporal permutation as discussed in Proposition 5. However in practice, the computation of addresses is simpler, and can be performed using XOR gates, thus avoiding the use of additional ROMs to store them.

2.2.4 *General linear permutations*

In this section, we discuss the implementation of a general SLP $\pi(P)$ using the previous structures. This is done by decomposing P into spatial and temporal permutations, i.e., permutations of the form (12) and (14).

A first idea is to use one spatial and one temporal permutation. Indeed, if the block P_4 is invertible, Gauss elimination yields

$$P = \begin{pmatrix} I_t & \\ P_2 P_4^{-1} & P_1 + P_2 P_4^{-1} P_3 \end{pmatrix} \begin{pmatrix} P_4 & P_3 \\ & I_k \end{pmatrix}.$$

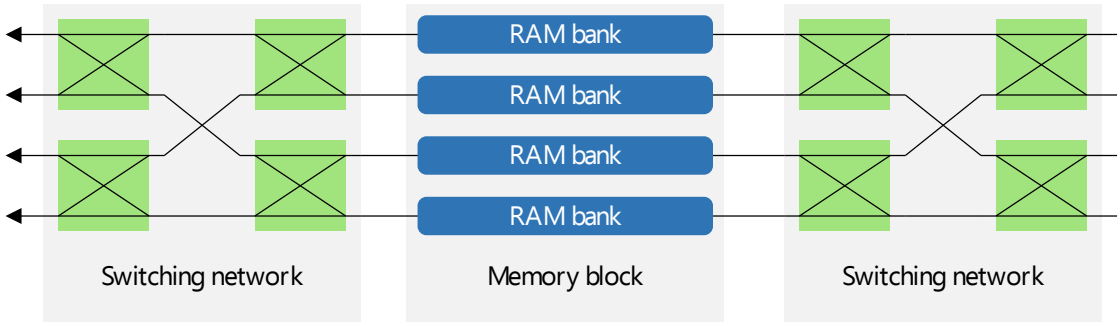
This means that $\pi(P)$ can be implemented using a memory block followed by an SNW. For the spatial part, $\operatorname{rank} P_2 P_4^{-1} = \operatorname{rank} P_2$, i.e., our implementation will have $\operatorname{rank} P_2 \cdot 2^{k-1}$ switches, which matches the lower bound of Corollary 2.

Analogously, if P_1 is invertible, it is possible to decompose an SLP using an SNW followed by a memory block:

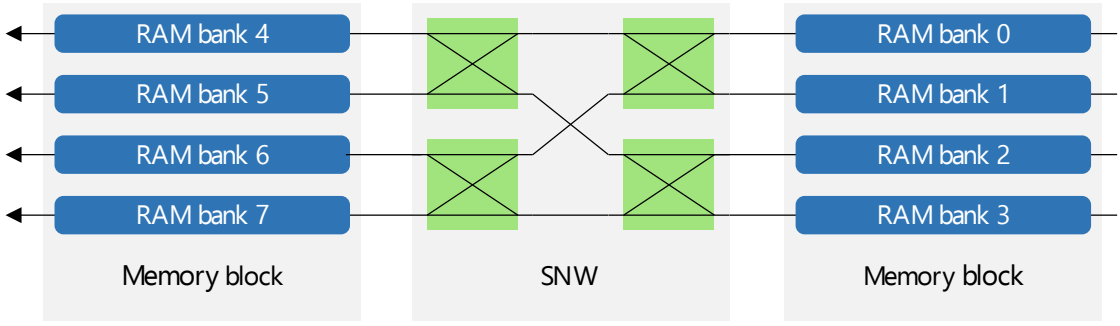
$$P = \begin{pmatrix} P_4 + P_3 P_1^{-1} P_2 & P_3 \\ & P_1 \end{pmatrix} \begin{pmatrix} I_t & \\ P_1^{-1} P_2 & I_k \end{pmatrix}. \tag{15}$$

Again, the construction will be optimal.

However, if neither P_1 nor P_4 are invertible, none of the solutions above exist. Hence, three blocks are needed and two possibilities exist, depicted in Fig. 7: the SNW-RAM-SNW structure (Section 2.2.4), and the RAM-SNW-RAM structure (Section 2.2.4).



(a) SNW-RAM-SNW



(b) RAM-SNW-RAM

Figure 7: Two possible architectures for a streaming permutation.

SNW-RAM-SNW. An SNW-RAM-SNW implementation (Fig. 7a) corresponds to the factorization

$$P = \begin{pmatrix} I_t & \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} M_4 & M_3 \\ & M_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R_2 & R_1 \end{pmatrix}. \tag{16}$$

Using our method of implementation, the number of switches involved equals $(\operatorname{rank} L_2 + \operatorname{rank} R_2)2^{k-1}$. Thus we want to minimize $\operatorname{rank} L_2 + \operatorname{rank} R_2$ for an optimal implementation. This decomposition is studied in Chapter 6, summarized in the following theorem:

Theorem 2. *If* P *is an invertible* $n \times n$ *matrix, then* (16) *satisfies:*

$$\operatorname{rank} L_2 + \operatorname{rank} R_2 \geqslant \max(\operatorname{rank} P_2, n - \operatorname{rank} P_4 - \operatorname{rank} P_1), \text{ and}$$
$$\operatorname{rank} M_3 = \operatorname{rank} P_3.$$

Further, there exists a decomposition (16) *reaching this bound.*

This theorem yields the minimal number of switches possible for the assumed architecture SNW-RAM-SNW:

$$\max(\operatorname{rank} P_2, n - \operatorname{rank} P_4 - \operatorname{rank} P_1) \cdot 2^{k-1}, \tag{17}$$

along with the existence of a solution reaching this bound. An algorithm to compute this solution in cubic arithmetic time in n is provided in Chapter 6[11].

11 The "rank exchange" section in Chapter 6 can be used in some cases to balance the ranks of L_2 and R_2. For instance, if $\operatorname{rank} L_2$ and $\operatorname{rank} R_2$ are both odd, it is interesting to reduce the rank of L_2 by one and increase the rank of R_2 by one, thus making them both even, and therefore easier to implement using 4-input multiplexers.

If $\operatorname{rank} P_2 \geqslant n - \operatorname{rank} P_1 - \operatorname{rank} P_4$, then Theorem 2 yields a routing-optimal solution. However, if $\operatorname{rank} P_2 < n - \operatorname{rank} P_1 - \operatorname{rank} P_4$, the solution has more switches than suggested by Corollary 2 (which does not fix the architecture). It turns out that in this case the next architecture is optimal in terms of the number of switches, at the price of twice the RAM.

RAM-SNW-RAM. A RAM-SNW-RAM implementation (Fig. 7b) corresponds to the factorization

$$P = \begin{pmatrix} L_4 & L_3 \\ & L_1 \end{pmatrix} \begin{pmatrix} I_t & \\ M_2 & M_1 \end{pmatrix} \begin{pmatrix} R_4 & R_3 \\ & R_1 \end{pmatrix}. \tag{18}$$

A switching-optimal solution is guaranteed by the following corollary:

Corollary 4. *If P is an invertible $n \times n$ matrix, there exists a decomposition (18) that satisfies $\operatorname{rank} M_2 = \operatorname{rank} P_2$.*

Proof. We use Theorem 2 on the transpose of P:

$$P^T = \begin{pmatrix} I_t & \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} M_4 & M_3 \\ & M_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R_2 & R_1 \end{pmatrix}.$$

A computation then shows that

$$P = \begin{pmatrix} M_4^T & R_2^T \\ & R_1^T \end{pmatrix} \begin{pmatrix} I_t & \\ M_3^T & M_1^T \end{pmatrix} \begin{pmatrix} I_t & L_2^T \\ & L_1^T \end{pmatrix},$$

which has the expected form, with $\operatorname{rank} M_3^T = \operatorname{rank} M_3 = \operatorname{rank} P_2^T = \operatorname{rank} P_2$. □

In summary, the RAM-SNW-RAM solution is always optimal in terms of the number of switches. However, if $\operatorname{rank} P_4 + \operatorname{rank} P_2 + \operatorname{rank} P_1 \geqslant n$, SNW-RAM-SNW offers a better solution with half the RAM.

Bit-permutations. As bit-permutations are a special case of linear permutations, our technique yields optimal implementations for them as well. As we have in this case $2 \cdot \operatorname{rank} P_2 = n - \operatorname{rank} P_1 - \operatorname{rank} P_4$, this means that an implementation (of a non-spatial streaming bit-permutation) uses either

- $2^k \cdot \operatorname{rank} P_2$ switches and 2^k banks of RAM, or
- $2^{k-1} \cdot \operatorname{rank} P_2$ switches and 2^{k+1} banks of RAM.

Note that in this case, $\operatorname{rank} P_2$ is the number of 1s that appear in the submatrix P_2. The decomposition proposed in [63] was already optimal in this particular case, and yields the case with $2^k \cdot \operatorname{rank} P_2$ switches and 2^k banks of RAM.

2.2.5 *Examples*

We now illustrate our method using three important examples of SLPs; the bit-reversal, the perfect-shuffle, and the half-reversal.

Bit-reversal. We consider the implementation of the bit-reversal permutation $(\pi(J_n), 2^k)$. Blocking the matrix J_n in the form (9) yields

$$J_n = \left(\begin{array}{c|c} & J_t \\ \hline & J_{k-t} \\ J_t & \end{array}\right) \text{ if } t \leqslant k, \text{ and } J_n = \left(\begin{array}{c|c} & J_k \\ \hline J_{t-k} & \\ \hline J_k & \end{array}\right) \text{ otherwise.}$$

Since in both cases $\operatorname{rank} P_2 = \min(t,k)$, Corollary 2 states that at least $S_{\pi(J_n),2^k}/2 = \min(t,k)\cdot 2^{k-1}$ switches are needed. However, Theorem 2 shows that an SNW-RAM-SNW structure requires twice this amount: $\min(t,k)\cdot 2^k$ switches, based on, for example,

$$J_n = \left(\begin{array}{c|c} I_t & \\ \hline & \begin{array}{cc} I_{k-t} & \\ & I_t \end{array} \\ \hline J_t & \end{array}\right)\left(\begin{array}{c|c} I_t & J_t \\ \hline & \begin{array}{cc} J_{k-t} & \\ & I_t \end{array} \end{array}\right)\left(\begin{array}{c|c} I_t & \\ \hline & \begin{array}{cc} I_{k-t} & \\ & I_t \end{array} \\ \hline J_t & \end{array}\right),$$

for the case $t \leqslant k$, and

$$J_n = \left(\begin{array}{cc|c} I_k & & \\ & I_{t-k} & \\ \hline J_k & & I_k \end{array}\right)\left(\begin{array}{cc|c} I_k & & J_k \\ & J_{t-k} & \\ \hline & & I_k \end{array}\right)\left(\begin{array}{cc|c} I_k & & \\ & I_{t-k} & \\ \hline J_k & & I_k \end{array}\right)$$

otherwise. If, on the other hand, we choose a RAM-SNW-RAM structure, we can reach the minimal number of switches with, for example, for $t \leqslant k$,

$$J_n = \left(\begin{array}{c|c} I_t & J_t \\ \hline & \begin{array}{cc} I_{k-t} & \\ & I_t \end{array} \end{array}\right)\left(\begin{array}{c|c} I_t & \\ \hline & \begin{array}{cc} J_{k-t} & \\ J_t & I_t \end{array} \end{array}\right)\left(\begin{array}{c|c} I_t & J_t \\ \hline & \begin{array}{cc} I_{k-t} & \\ & I_t \end{array} \end{array}\right),$$

and otherwise

$$J_n = \left(\begin{array}{cc|c} I_k & & J_k \\ & I_{t-k} & \\ \hline & & I_k \end{array}\right)\left(\begin{array}{cc|c} I_k & & \\ & J_{t-k} & \\ \hline J_k & & I_k \end{array}\right)\left(\begin{array}{cc|c} I_k & & J_k \\ & I_{t-k} & \\ \hline & & I_k \end{array}\right).$$

Note in all cases the simplicity of the control logic: only a t-bit counter and t inverters are needed.

However, routing optimality comes at the price of memory optimality. A computation shows that the bit-reversal satisfies

$$\delta_{\pi(J_n),2^k} = 2^t - a_{t-k},$$

where

$$a_i = \begin{cases} 1 & \text{if } i \leqslant 0, \\ 2 & \text{if } i = 1 \text{ and} \\ 1 + 2a_{i-2} & \text{otherwise.} \end{cases}$$

The SNW/RAM/SNW architecture has a latency of $\delta_{\pi(J_n),2^k}$ cycles, and uses 2^k RAM banks that have a size of $\delta_{\pi(J_n),2^k}$ words each. The RAM/SNW/RAM architecture uses twice the number of RAM banks (with the same capacity), and has a latency doubled.

Perfect-shuffle. We consider the streaming perfect-shuffle $(\pi(C_n), 2^k)$. When blocked as in (9), it satisfies

$$P_2 = E^{1,t}_{k,t},$$

where $E^{i,j}_{k,t}$ is the $k \times t$ matrix containing a 1 at the i^{th} row and the j^{th} column, and 0s elsewhere. According to Corollary 2, it requires $S_{\pi(J_n),2^k}/2 = 2^{k-1}$ 2×2-switches for its implementation. Additionally, the minimal latency for this SLP is $\delta_{\pi(J_n),2^k} = 2^{t-1}$ cycles. The method described in Theorem 2 yields the decomposition

$$C_n = \begin{pmatrix} I_t & \\ E^{k,t}_{k,t} & I_k \end{pmatrix} \begin{pmatrix} C_t & E^{t,1}_{t,k} \\ & C_k \end{pmatrix} \begin{pmatrix} I_t & \\ E^{1,1}_{k,t} & I_k \end{pmatrix}.$$

The corresponding implementation is memory optimal, but uses 2^k switches. Conversely, the RAM/SNW/RAM implementation yielded by the decomposition

$$C_n = \begin{pmatrix} C_t & E^{t,k}_{t,k} \\ & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ E^{k,1}_{k,t} & C_k \end{pmatrix} \begin{pmatrix} I_t & E^{1,1}_{t,k} \\ & I_k \end{pmatrix}$$

is routing optimal, but has a latency of 2^t cycles, and uses a total memory capacity of 2^n words.

Half-reversal. We consider the implementation of a streaming half-reversal permutation $(I_{2^{n-1}} \oplus J_{2^{n-1}}, 2^k) = (\pi(V_n), 2^k)$. In this case, a block-LU decomposition (15) is possible, as P_1 is invertible:

$$V_n = (V_t \oplus I_k) \cdot \begin{pmatrix} I_t & \\ F_{k,t} & I_k \end{pmatrix},$$

where $F_{k,t}$ is the the $k \times t$ matrix containing 1s in the first column, and 0s elsewhere. This yields a RAM/SNW implementation that is both memory and routing optimal.

2.3 RESULTS

We evaluate our method in two ways. First, we consider one particular, but important example: the streamed bit reversal. We compare our two proposed architectures (one of which is optimal) against a prior solution. Second, we compare our streamed permutations against prior solutions that we found in the literature. We show a table summarizing the similarities and differences and illustrate these with three example settings.

Fig. 8 shows throughput versus area for a bit reversal on 2^{11} 16-bit elements for the two different architectures implemented with $k \in \{1, \ldots, 5\}$, i.e., 2 to 32 ports, and $t = 11 - k$. In this case, our SNW-RAM-SNW solution is equal to the one proposed by [63]. For each of the two solutions we also implemented the FPGA-specific optimization that uses 4-input multiplexers as sketched in Fig. 6, which yields significant area gains.

We compare against the RAM-SNW-RAM solution in [45], which is more general in that it can handle (fixed) arbitrary, also non-linear permutations. The target is a Virtex-7 xc7vx1140tflgl1930 FPGA, using Xilinx Vivado 2014.4.

Comparison against prior work. Table 2 summarizes the similarities and differences between our solutions (SNW-RAM-SNW and RAM-SNW-RAM) and prior works. As the table shows, only ours provide guaranteed optimal switching complexity at similar RAM cost for linear permutations.

To show the difference with an example, Fig. 9 compares, for different streaming scenarios, the number of switches used by the different architectures. In (a) all specified SLPs are considered, in (b) and (c), the full number is too large and we chose

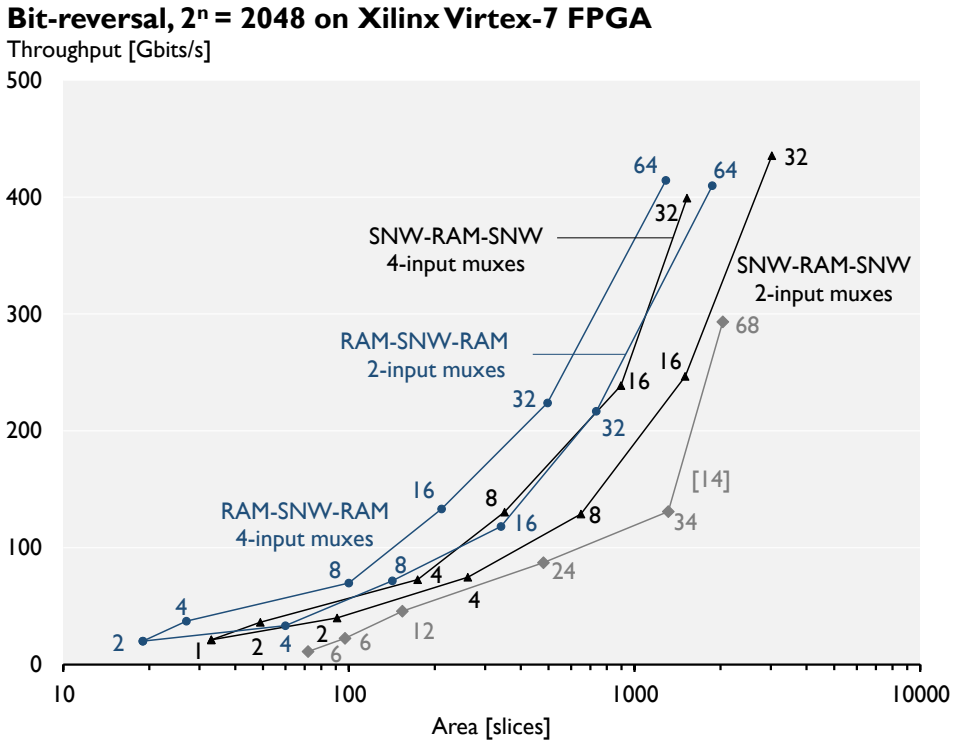
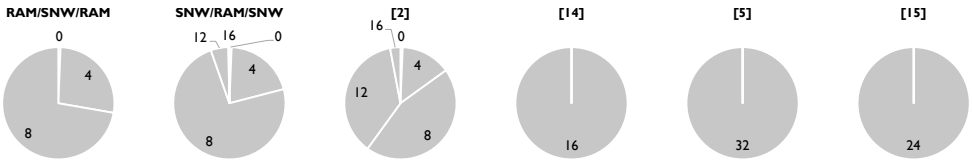
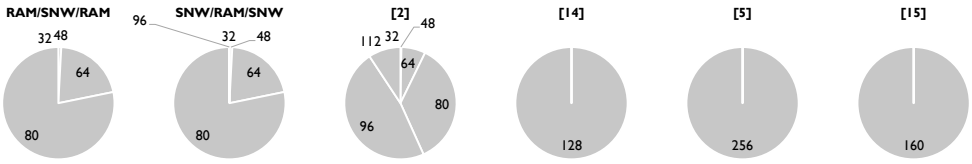


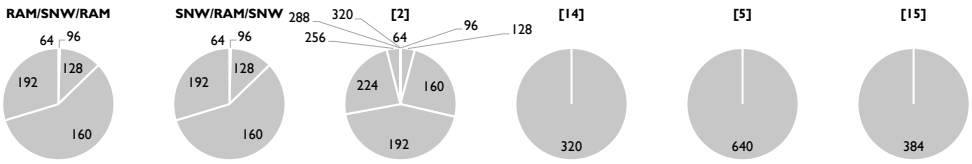
Figure 8: Comparison of our two structures for a bit-reversal permutation on 2048 16-bit elements for different multiplexer sizes vs. [45]. Labels: number of BRAM tiles. In this example, the SNW-RAM-SNW structure that uses 2-input muxes is equivalent to [63].



(a) Permutations of $2^n = 32$ elements with $2^k = 8$ ports



(b) Permutations of $2^n = 4096$ elements with $2^k = 32$ ports



(c) Permutations of $2^n = 4096$ elements with $2^k = 64$ ports

Figure 9: Number of switches needed for 10^7 random SLPs with different architectures.

	Type	Number of RAM banks	Total memory capacity	Latency	2-input multiplexers
Any permutation (σ_N, K)					
[45]	RAM	$2 \cdot K$	$4 \cdot N$	$2 \cdot N/K$	$2 + (\lceil \log_2 K \rceil - 1) \cdot 2^{\lceil \log_2 K \rceil + 1}$
[12]	RAM	$2^{\lceil \log_2 K \rceil}$	$2^{\lceil \log_2 K \rceil} + \lceil \log_2 (N/K) \rceil$	$2^{\lceil \log_2 (N/K) \rceil}$	$4 + (\lceil \log_2 K \rceil - 1) \cdot 2^{\lceil \log_2 K \rceil + 2}$
[10]	RAM	$2^{\lceil \log_2 K \rceil}$	$2^{\lceil \log_2 K \rceil} + \lceil \log_2 (N/K) \rceil$	$2^{\lceil \log_2 (N/K) \rceil}$	$\lceil \log_2 K \rceil \cdot 2^{\lceil \log_2 K \rceil + 1}$
[38]	RAM	$\mathbf{K}$	$\boldsymbol{\delta_{\sigma_N, K} \cdot K}$	$\boldsymbol{\delta_{\sigma_N, K}}$	$4 + (\lceil \log_2 K \rceil - 1) \cdot 2^{\lceil \log_2 K \rceil + 2}$
Linear permutations $(\pi(P), 2^k)$					
[45]	RAM	2^{k+1}	2^{n+2}	2^t	$2 + (k-1) \cdot 2^{k+1}$
[12]	RAM	2^k	2^n	2^t	$4 + (k-1) \cdot 2^{k+2}$
[10]	RAM	2^k	2^n	2^t	$k \cdot 2^{k+1}$
[38]	RAM	2^k	$\boldsymbol{\delta_{\pi(P), 2^k} \cdot 2^k}$	$\boldsymbol{\delta_{\pi(P), 2^k}}$	$4 + (k-1) \cdot 2^{k+2}$
[63]	RAM	2^k	2^{n+1}	2^t	$\geqslant \max(\operatorname{rank} P_2, n - \operatorname{rank} P_4 - \operatorname{rank} P_1) \cdot 2^k$
SRS	RAM	2^k	$\boldsymbol{\delta_{\pi(P), 2^k} \cdot 2^k}$	$\boldsymbol{\delta_{\pi(P), 2^k}}$	$\max(\operatorname{rank} P_2, n - \operatorname{rank} P_4 - \operatorname{rank} P_1) \cdot 2^k$
RSR	RAM	2^{k+1}	$\geqslant \delta_{\pi(P), 2^k} \cdot 2^k$	$\geqslant \delta_{\pi(P), 2^k}$	$\mathbf{rank\, P_2 \cdot 2^k}$
Bit-reversal $(\pi(J_n), 2^k)$					
[45]	RAM	2^{k+1}	2^{n+2}	2^t	$2 + (k-1) \cdot 2^{k+1}$
[12]	RAM	2^k	2^n	2^t	$4 + (k-1) \cdot 2^{k+2}$
[10]	RAM	2^k	2^n	2^t	$k \cdot 2^{k+1}$
[11]	RAM	2^k	2^n	2^t	$k \cdot 2^{k+1}$
[38]	RAM	2^k	$\boldsymbol{\delta_{\pi(J_n), K} \cdot 2^k}$	$\boldsymbol{\delta_{\pi(J_n), K}}$	$4 + (k-1) \cdot 2^{k+2}$
[63]	RAM	2^k	2^{n+1}	2^t	$\min(t, k) \cdot 2^{k+1}$
	FIFO	$\begin{cases} t \cdot 2^k & \text{if } t \leqslant k, \\ (k+1) \cdot 2^k & \text{otherwise} \end{cases}$	$\begin{cases} \boldsymbol{2^n - 2^k} & \text{if } t \leqslant k, \\ 2^n & \text{otherwise} \end{cases}$	$\begin{cases} \boldsymbol{2^t - 1} & \text{if } t \leqslant k, \\ 2^t & \text{otherwise} \end{cases}$	
[23]	and				$\boldsymbol{\min(t, k) \cdot 2^k}$
	RAM				
SRS	RAM	2^k	$\boldsymbol{\delta_{\pi(J_n), K} \cdot 2^k}$	$\boldsymbol{\delta_{\pi(J_n), K}}$	$\min(t, k) \cdot 2^{k+1}$
RSR	RAM	2^{k+1}	$\delta_{\pi(J_n), K} \cdot 2^{k+1}$	$2 \cdot \delta_{\pi(J_n), K}$	$\boldsymbol{\min(t, k) \cdot 2^k}$
Perfect shuffle $(\pi(C_n), 2^k)$					
[45]	RAM	2^{k+1}	2^{n+2}	2^t	$2 + (k-1) \cdot 2^{k+1}$
[12]	RAM	2^k	2^n	2^t	$4 + (k-1) \cdot 2^{k+2}$
[10]	RAM	2^k	2^n	2^t	$k \cdot 2^{k+1}$
[38]	RAM	2^k	$\boldsymbol{2^n - 1}$	$\boldsymbol{2^t - 1}$	$4 + (k-1) \cdot 2^{k+2}$
[63]	RAM	2^k	2^{n+1}	2^t	2^{k+1}
SRS	RAM	2^k	$\boldsymbol{2^n - 1}$	$\boldsymbol{2^t - 1}$	2^{k+1}
RSR	RAM	2^{k+1}	2^n	2^t	$\boldsymbol{2^k}$

Table 2: Comparison of different architectures using RAMs, in the case of a full-throughput SLP. Values in bold are optimal ones.

10^7 random samples instead. The pie charts show the distribution of the number of switches needed for these SLPs. As shown in this chapter, one of our solutions (the two leftmost in the table) always minimizes the number of switches needed. We observe the improvement over prior work and also that for larger scenarios, most of the permutations can be implemented optimally using SNW-RAM-SNW. As we have seen, this is not true for the bit-reversal.

2.4 RELATED WORK

Switching networks for sets of permutations. Switching networks that can execute all permutations (in a non-streamed way) are a classic topic in computer science [6, 84]. A variant of this problem occurred in Section 2.2.3 where we implemented streamed spatial permutations. Namely, we had to build a minimal switching network capable of passing a subset of permutations.[12] Our solution was based on a reduced Omega network and we proved optimality. The complete Omega network has been heavily studied in [61, 40, 78, 49]. Beyond that, the problem of finding a minimal switching network to perform a given set of permutations appears to have not received much attention in the literature. An exception is the last section in [78], which, however, produces only upper bounds for few cases.

SNW-RAM-SNW structure. We now restrict ourselves to the structure proposed in Section 2.2.4. This architecture has already been proposed for streamed linear permutations in [63], which also proves optimality for bit-permutations. For these permutations, our solution is equal (Fig. 8 shows one example).

However, [63] has two shortcomings that we resolve here. First, the method to derive an SNW-RAM-SNW implementation is in general not optimal (see Fig. 9). Second, [63] does not consider the alternative architecture RAM-SNW-RAM, which, in some cases provides solutions with fewer switches at the cost of twice the RAM. In this chapter we resolve both problems completely by establishing an architecture-oblivious sharp lower bound for the number of switches needed and a technique for obtaining that optimal solution using the SNW-RAM-SNW or RAM-SNW-RAM architecture. We precisely characterize the cases where the latter wins.

As a minor point, the solution in [63] uses a double-buffering method to achieve full-throughput (as they mention a memory requirement of 2^{n+1} words in the last section). We propose an alternative method in Proposition 5 that does not require additional RAM capacity.

This SNW-RAM-SNW architecture has also been used in [12] to implement the streaming permutations needed in a bitonic sorting network (which are all linear). They achieve an efficient memory usage, but the method used (folding a Clos permutation network) doesn't harness the specificity of the particular permutations they consider, and the resulting design requires two complete switching networks (that allow any permutation), which also makes the control logic much more complex.

Similarly, [10] offers a solution based on a Beneš network to build a streamed solution for any, also non-linear, given permutation. However, the method proposed only works on sizes that are a power of two ($N = 2^n$), and for power-of-2 streaming width ($K = 2^k$). Because it is more general, it is not optimal for the linear case. Additionally, the generated datapath is independent of the desired permutation. The control logic is

12 Specifically a coset Hg, where g is a linear permutation, and H a subgroup of bit complement permutations, i.e., permutations that map an index i to $i_b + v$, where v is a given bit vector.

also more complex, as it uses ROM look-up tables to store memory addresses and the control bit of every switches for every cycles. This allows flexibility in the sense that different permutations can be implemented simply by modifying these tables, but is clearly suboptimal for a single fixed permutation. In Fig. 9, we showed how our solutions outperform this method. The same structure was used for the specific case of the bit-reversal in [11].

RAM-SNW-RAM structure. The RAM-SNW-RAM structure was considered in [45] to implement any (including non-linear) streaming permutation of any size. A shortcoming is that the central SNW has to be able to pass any spatial permutation. Further, it considers only double-buffering for its temporal permutations. We compared our different architectures in Fig. 8 and 9.

Other architectures for streamed permutations. Other approaches for building a fixed permutation technique include [59], which proposes a register based implementation, and [31], which is specific to implementing stride permutations. These two methods have in common that they use registers to delay elements. In this chapter we choose a more regular architecture using RAM banks instead, which are available on FPGAs, to spare logic. A mix of FIFOs and RAMs is used in [23] to implement the bit-reversal. It achieves a routing optimality in any case, and is optimal in latency and in memory capacity if $t < k$. The implementation requires the memory to be split in more banks than 2^k, but is a good alternative on architectures where large RAMs are not advantageous (ASICs).

2.5 CONCLUSION

We introduced a novel measurement on streaming permutation, the *routing entropy*, that allows to quantify its routing complexity. We proved that it constitutes a lower bound on the number of 2-input multiplexers that any of its implementation requires. In the case of linear permutations, we provided a constructive method (the RAM/SNW/RAM architecture) that achieves this lower bound, thus establishing their exact switching complexity. A remaining question is whether this bound is sharp for any streaming permutation.

We provided a second method, the SNW/RAM/SNW architecture, that guaranties memory optimality, and that yields a number of multiplexers that is optimal for this architecture. In some cases however (that we precisely characterize), this number is higher than the global optimum.

The underlying key idea of these two methods was to phrase the problem as a specific matrix factorization and apply techniques from linear algebra to construct solutions and prove their optimality. A topic for future work is to search for architectures that would be able to achieve memory and routing optimality at the same time, or to characterize the trade-off that might appear.

3

MEMORY-EFFICIENT FAST FOURIER TRANSFORM ON STREAMING DATA BY FUSING PERMUTATIONS

In this chapter, published in [73], we extend the prior work corresponding to the designs shown in Fig. 2 with a novel method that can reduce memory requirement by roughly one half. The method is more generally applicable beyond FFTs: it designs a circuit that can perform a small number of data permutations, which take the input streamed over several cycles.

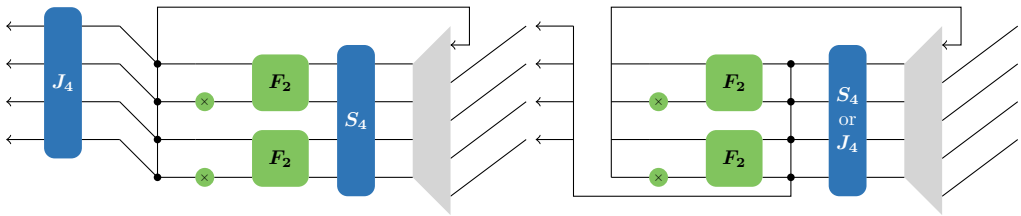
The architecture we propose is shown in Fig. 10b and fuses the hardware performing the two permutations in Fig. 10a to reduce cost, and in particular the memory required. As a result, a memory-efficient Pease FFT can be implemented for any radix. Here, we focus on radix 2.The method for fusing the streaming permutation is more generally applicable but the FFT was the motivation for this work.

Contributions. Our main contributions are as follows:

- We present a method to design a specialized datapath that can realize a given (small) set of permutations, taking the input streamed over several cycles. This datapath is cheaper than one capable of performing all permutations. The method is limited to the class of *linear* permutations, which contains bit reversal, perfect shuffle, matrix transpositions, and permutations needed in other FFTs beyond Pease, sorting networks, Viterbi decoders, filter banks, and other algorithms. The datapath we design consists of basic logic and RAMs, which is well-suited for implementation on FPGAs.
- As a major application, we propose a novel variant of a streamed FFT architecture (as shown in Fig. 10b) that reduces the RAMs required by prior work by up to one half.

3.1 STREAMING MULTIPLE LINEAR PERMUTATIONS

The prior techniques from Chapter 2 are sufficient to implement the streaming permutations, and thus the streaming FFTs shown in Fig. 2. However, they cannot be used for



(a) Iterative and streaming reuse. [47]

(b) Iterative and streaming reuse with fused permutations.

Figure 10: Our contribution: streaming design with iterative reuse with two permutations fused.

the structure in Fig. 10b, where the same datapath has to handle two different SLPs[1]. An immediate solution would be to use general streaming permutation methods, like [45] or [38]. They propose, respectively, a structure as in Fig. 7a and 7b, but replace the specialized SNWs by complete, and thus more expensive permutation networks. In this section, we propose a method to implement in hardware a datapath capable of rearranging streaming data according to a small number of given linear permutations, thus reducing the implementation cost compared to a general solution.

Problem statement. Formally, we are given a list

$$\pi(P^{(0)}), \pi(P^{(1)}), \ldots, \pi(P^{(s-1)})$$

of linear permutations, and a streaming width 2^k. Our goal is to implement an architecture that performs the permutation $\pi(P^{(i)})$ on the i^{th} dataset, streamed over 2^k ports.

The main idea first decomposes each permutation as in Theorem 2, i.e., for all $0 \leqslant i < s$,

$$\pi(P^{(i)}) = \pi(L^{(i)}) \cdot \pi(C^{(i)}) \cdot \pi(R^{(i)}),$$

where each factor is either temporal or spatial. The global architecture can then be implemented by a sequence of blocks that each perform either a sequence of temporal or a sequence of spatial SLPs. We now consider these two cases and describe their implementation.

3.1.1 *Sequence of Spatial SLPs*

We assume a given list of bit matrices $P^{(0)}, P^{(1)}, \ldots, P^{(s-1)}$, such that all $\pi(P^{(i)})$ are spatial, i.e.,

$$P^{(i)} = \begin{pmatrix} I_t & \\ P_2^{(i)} & P_1^{(i)} \end{pmatrix}, \quad 0 \leqslant i < s. \tag{19}$$

We first design solutions for two special cases from which we then build the solution for the general case.

Multiplexer array. We first consider the case where all the SLPs $\pi(P^{(0)}), \ldots, \pi(P^{(s-1)})$ are steady spatial permutations, i.e., for every i,

$$P^{(i)} = I_t \oplus P_1^{(i)} = \begin{pmatrix} I_t & \\ & P_1^{(i)} \end{pmatrix}. \tag{20}$$

Since each such SLP is a different wiring, the list of those can be implemented with an array of 2^k d-input multiplexers, where d is the number of unique matrices in the list (see Fig. 11b).

For instance, if the list contains two different matrices $I_t \oplus A$ and $I_t \oplus B$, it is possible to implement both using a structure where each output port p is the output of a multiplexer connected to the inputs $A^{-1}p_b$ and $B^{-1}p_b$, and controlled by a counter. Of course, the multiplexers connected twice to the same input can be simplified to a simple wire, leaving an actual implementation consisting of

$$\left|\{p \mid A^{-1}p_b \neq B^{-1}p_b\}\right| = 2^k - 2^{k-\operatorname{rank}(A^{-1}+B^{-1})}$$

1 The direct sum, i.e. block diagonal composition of two linear permutations is in general not a linear permutation.

2-input multiplexers. If $A = B$, then $A^{-1} = B^{-1}$ and thus the sum is 0 (since addition is modulo 2), which means the implementation consists only of wires, as expected.

Switching array. We now consider another special case where, for every i, $P_1^{(i)} = I_k$ and all elements of $P_2^{(i)}$ are zero, except for its last row, which we denote with v_i^T. Formally, for every i,

$$P^{(i)} = \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_i^T & & & 1 \end{pmatrix}. \tag{21}$$

In this case, (10) shows that $\pi(P^{(i)})$ is the permutation that exchanges each pair within a chunk of 2^k elements, every time the corresponding cycle c is such that $c_b \cdot v_{ib} = 1$.

Therefore, P can be implemented using an array of 2^{k-1} 2×2-switches. All these switches are controlled by the output of a single s-input multiplexer that chooses among the results of the scalar products $c_b \cdot v_{ib}$, for $0 \leqslant i < s$. These scalar products are computed using XOR gates on a timer c_b.

Fig. 11c shows such a switching array that can implement any spatial $P^{(i)}$ in (21) for $k = 2$.

General Spatial SLP. We return now to the general case (19), which we will decompose into matrices of the form (20) and (21) to implement it with the previous structures. We first consider the matrix M of size $k \times st$ that concatenates the matrices $P_2^{(i)}$:

$$M = \begin{pmatrix} P_2^{(0)} & P_2^{(1)} & \cdots & P_2^{(s-1)} \end{pmatrix}.$$

Using Gaussian elimination, it is possible to find an invertible matrix K of size $k \times k$ such that KM has $m = \operatorname{rank} M$ non-zero rows at the top. This implies that for every i the matrix $KP_2^{(i)}$ has the form

$$KP_2^{(i)} = \begin{pmatrix} v_{1,i}^T \\ v_{2,i}^T \\ \vdots \\ v_{m,i}^T \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where the m top rows are denoted with $v_{j,i}^T$. Note that some of these may be zero for a given i. Direct computation yields now the decomposition into the prior special cases:

$$P^{(i)} = \begin{pmatrix} I_t & \\ & K^{-1}S_k^{k-m} \end{pmatrix} \cdot \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_{1,i}^T & & & 1 \end{pmatrix} \begin{pmatrix} I_t & \\ & S_k \end{pmatrix} \cdot \\ \vdots \\ \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_{m,i}^T & & & 1 \end{pmatrix} \begin{pmatrix} I_t & \\ & S_k \end{pmatrix} \cdot \\ \begin{pmatrix} I_t & \\ & KP_1^{(i)} \end{pmatrix}.$$

The corresponding architecture can now be read off from right to left:

1. a multiplexer array that permutes the wires as $\pi(I_t \oplus KP_1^{(i)})$ for the i^{th} dataset,
2. a sequence of m switching arrays, parameterized, respectively, by $v_m, v_{m-1}, \ldots, v_1$, each preceded by a perfect shuffle of the wires, and
3. a rewiring performing the permutation $\pi(I_t \oplus K^{-1}S_k^{k-m})$.

Cost. The structure that we derived consists of rank M arrays of 2^{k-1} switches each, and one array of at most 2^k multiplexers.

3.1.2 *Sequence of Temporal SLPs*

We assume a given list of bit matrices $P^{(0)}, P^{(1)}, \ldots, P^{(s-1)}$, such that for $0 \leqslant i < s, \pi(P^{(i)})$ is temporal, i.e.,

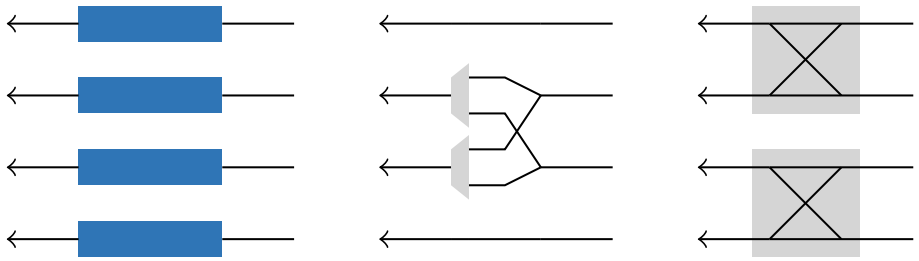
$$P^{(i)} = \begin{pmatrix} P_4^{(i)} & P_3^{(i)} \\ & P_1^{(i)} \end{pmatrix} = \begin{pmatrix} I_t & \\ & P_1^{(i)} \end{pmatrix} \cdot \begin{pmatrix} P_4^{(i)} & P_3^{(i)} \\ & I_k \end{pmatrix}.$$

This decomposition yields a sequence of steady spatial SLPs that can be implemented using an array of switches a explained earlier, and a *RAM array*:

RAM array. A structure that permutes a dataset i according to $\pi(P^{(i)})$, where

$$P^{(i)} = \begin{pmatrix} P_4^{(i)} & P_3^{(i)} \\ & I_t \end{pmatrix}$$

can be implemented using an array of 2^k dual-ported RAM banks of 2^t words. Each of these banks have a write port connected to one of the inputs of the block, and a read port connected to the corresponding output (see Fig. 11a). The write and read addresses ensure that data are correctly permuted (accordingly to (11)), and are respectively controlled using two t-bits timers: c, that starts when a new dataset arrives, and c', that starts when the output begins.



(a) RAM banks array (b) Multiplexer array (c) Switching array

Figure 11: The basic blocks we use, here for a streaming width of $2^k = 4$. (a) can pass any temporal permutation; (b) implements the two spatial steady SLPs $\pi(I_t \oplus J_2)$ and $\pi(I_n)$; (c) implements (21).

Latency. The output begins as early as possible, to minimize the latency, for each different permutation. Therefore, c' is triggered when c reaches the value corresponding to the maximal lifetime δ_i of an element in the permutation:

$$\delta_i = \max_{p,c}(c - d(i,p,c)), \quad \text{with } d(i,p,c)_b = P_4^{(i)} c_b + P_3^{(i)} p_b.$$

Conflict-free addressing. Besides permuting correctly the data, the read and write addresses need to ensure a conflict-free access. This means that an incoming element must not be written to a place where an element of a previous dataset has not been read yet. One solution is to use double buffering [45, 63], but this requires doubling the size of each RAM bank.

The solution we propose is to always write an element of a dataset where the same element of the previous dataset was read. Namely, for the p^{th} port, the first dataset received is written consecutively in the bank, i.e., at address c_b. It is then read at the address $(P_4^{(0)})^{-1} c'_b + (P_4^{(0)})^{-1} P_3^{(0)} p_b$, to perform the first permutation $\pi(P^{(0)})$. Then, the second dataset is written where the first dataset was read to avoid conflicts, so at the address $(P^{(0)})_4^{-1} c_b + (P^{(0)})_4^{-1} P_3^{(0)} p_b$. It is then read at the address

$$(P_4^{(0)} P_4^{(1)})^{-1} c'_b + (P_4^{(0)} P_4^{(1)})^{-1} P_3^{(1)} p_b + (P_4^{(0)})^{-1} P_3^{(0)} p_b.$$

More generally, the i^{th} dataset is written (resp. read) at the address $U_i c_b + u_{i,p}$, (resp. $U_{i+1} c'_b + u_{i+1,p}$), where U_i is such that

$$\begin{cases} U_{i+1} = U_i (P_4^{(i \bmod s)})^{-1}, \\ U_0 = I_t, \end{cases}$$

and u_i satisfies

$$\begin{cases} u_{i+1,p} = U_{i+1} P_3^{(i \bmod s)} p_b + u_{i,p}, \\ u_{0,p} = 0. \end{cases}$$

We store the values of (U_i) in a ROM, controlled by a counter. Using AND and XOR gates, the term $U_i c_b$ is computed once for all the banks. Then, this signal is XORed with $u_{i,p}$ for each bank p to obtain the write address. The number of terms of (U_i) stored in the ROM is the least that guarantees conflict-free access (this length is bounded by the period[2] of (U_i)) The read address is obtained similarly.

2 The period of (U_i) is sq, where q is the smallest positive integer such that $(P_4^{(s-1)} P_4^{(s-2)} \cdots P_4^{(0)})^{-q} = I_t$.

Alternative addressing. It is also possible to use the method proposed in Proposition 5 here, allowing to use banks of size $\max_i \delta_i$ words. However, in our application, this value is close to 2^t due to the bit reversal.

3.1.3 *General sequence of SLPs*

Now we consider the general case of arbitrary invertible bit matrices $P^{(0)}, P^{(1)}, \ldots, P^{(s-1)}$. Using Theorem 2, we get, for each i, the decomposition

$$P^{(i)} = \begin{pmatrix} I_t & \\ L^{(i)} & I_k \end{pmatrix} \begin{pmatrix} C_4^{(i)} & C_3^{(i)} \\ & C_1^{(i)} \end{pmatrix} \begin{pmatrix} I_t & \\ R^{(i)} & I_k \end{pmatrix}.$$

This decomposition yields two sequences of spatial permutations, and one of temporal permutations. These can be implemented in a straightforward way using the previous structures.

Cost. The resulting architecture consists of one multiplexer array (as both sequences of spatial SLPs do not require any) containing a maximum of $2^k - 1$ multiplexers, an array of 2^k RAM banks (except in the special case where all the SLPs are spatial), and $(\operatorname{rank} M_L + \operatorname{rank} M_R) \cdot 2^{k-1}$ 2×2-switches, where

$$M_L = \begin{pmatrix} L_2^{(0)} & L_2^{(1)} & \cdots & L_2^{(s-1)} \end{pmatrix}, \text{ and } M_R = \begin{pmatrix} R_2^{(0)} & R_2^{(1)} & \cdots & R_2^{(s-1)} \end{pmatrix}.$$

Optimality. The number of RAM banks and the RAM latency match the bounds given in Chapter 2, and are therefore optimal. The number of switches, in the general case, depends on the different degrees of freedom appearing in the decompositions[3] from Theorem 2, and no optimality can be claimed. Of course, if the sequence of SLPs only contains one unique SLP, the design we obtain only differs from Chapter 2 by rewirings, and it therefore inherits the optimality properties.

Example: Fusing perfect shuffle and bit reversal. As an example, we design the permutation block in Fig. 10b capable of passing a perfect shuffle $\pi(C_4)$, and a bit reversal $\pi(J_4)$. Using the formulas of Section 2.2.5, we get the following (spatial/temporal/spatial) decompositions for J_4 and C_4:

$$J_4 = \left(\begin{array}{ccc|c} 1 & & & \\ & 1 & & \\ \hline & 1 & 1 & \\ 1 & & & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & & & 1 \\ & 1 & 1 & \\ \hline & & 1 & \\ & & & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & & & \\ & 1 & & \\ \hline & 1 & 1 & \\ 1 & & & 1 \end{array}\right), \text{ and} \tag{22}$$

$$C_4 = \left(\begin{array}{ccc|c} 1 & & & \\ & 1 & & \\ \hline & & 1 & \\ & 1 & & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} & 1 & & \\ 1 & & 1 & \\ \hline & & & 1 \\ & 1 & & \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & & & \\ & 1 & & \\ \hline 1 & & 1 & \\ & & & 1 \end{array}\right). \tag{23}$$

3 More precisely, the decomposition in Theorem 2 is optimal for each permutation taken individually, but as we compute independently these decompositions for each permutation, there is no guarantee in general that the global sequence yields an optimal rank for M_L and M_R.

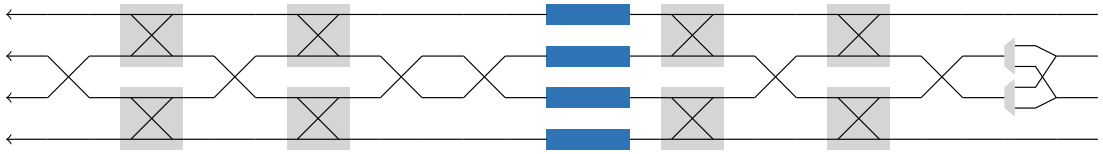


Figure 12: Datapath for the permutation block in Fig. 10b.

Figure 13: Datapath for a bit reversal on $2^n = 16$ elements streamed on $2^k = 4$ ports (see Section 2.2.4).

we derive a datapath that consists of two blocks that performs a sequence of spatial SLPs around a block that performs a sequence of temporal SLPs. For example, this sequence of temporal SLPs contains the two middle permutations in (23) and (22):

$$\pi \left(\left(\left[\begin{array}{cc|cc} & 1 & & \\ 1 & & 1 & \\ \hline & & & 1 \\ & & 1 & \end{array}\right] \right) \right) \quad \text{and} \quad \pi \left(\left(\left[\begin{array}{cc|cc} 1 & & & 1 \\ & 1 & 1 & \\ \hline & & 1 & \\ & & & 1 \end{array}\right] \right) \right).$$

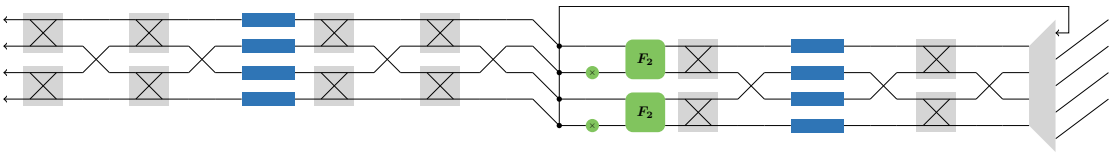
The resulting implementation consists of an array of two 2-input multiplexers, two stages of two 2×2-switches each, an array of four RAM banks, and two additional stages of two 2×2-switches each (Fig. 12). Compared to an architecture performing only the bit reversal derived in Section 2.2.4 (Fig. 13), it requires only two additional 2-input multiplexers.

More generally, an architecture that can stream both the bit reversal and the perfect shuffle on 2^n points with a streaming width 2^k differs from a bit-reversal-only datapath with the same architecture by only $2^k - 2$ 2-input multiplexers. In other words, the additional support for the perfect shuffle is obtained almost for free.

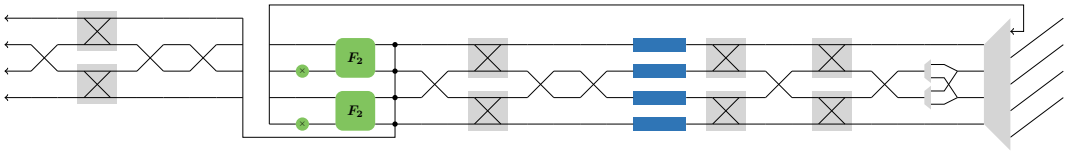
3.2 APPLICATION: PEASE FFT

To evaluate our fused permutation in a concrete case, we have built a generator (see Chapter 5) capable of producing designs as in Fig. 2 and 10b for Pease FFTs of arbitrary radix (Fig 2a shows the special case of radix 2). This generator takes as input the size 2^n of the FFT, the number of ports 2^k, the bit-width of the input data, and the desired radix 2^r, with $r|n$ and $r \leqslant k \leqslant n$. It outputs the corresponding design in the form of Verilog code. In this section, we briefly explain how this generator works.

Derivation of the FFT architecture. The generator first considers a Pease FFT algorithm of the corresponding radix, and the sequence of permutations that have to be supported by the permutation block of Fig. 10b. These are all linear, and the block is designed according to the techniques shown in Section 3.1. Butterflies and complex multipliers are then added to this permutation block within a loop, as in Fig. 10b. Some optimizations occur at this time. For instance, with a radix 2, during the implementation of the leftmost spatial permutation, it is possible to choose K such that



(a) Streaming iterative reuse as in Section 2.2.4



(b) Streaming iterative reuse with fused permutations

Figure 14: Radix-2 Pease FFT, iterative reuse with fused permutation $n = 4, k = 2$.

$v_{j,i} = 0$, for $i < n$ and $j < \min(t, k)$. Therefore, the $\min(t, k) - 1$ leftmost arrays of switches can safely be "unrolled," thus reducing the latency within the loop, and therefore improving the global throughput (see Fig. 14b). Only one stage of the leftmost spacial permutation remains in the loop.

Compared to the classic streaming reuse architecture (Fig. 14a), the design we obtain has an additional multiplexer stage and an additional switching array stage in the loop, but it does not have a dedicated structure to compute the bit reversal.

RTL graph. The design is then translated into an RTL graph, where additional optimizations are performed including the following:

- ROMs containing periodic values are simplified.
- ROMs containing a single value are replaced by a constant.
- Trivial arithmetic operations are simplified.
- A multiplexer with inputs coming from two multiplexers sharing the same inputs are fused into a single multiplexer.
- A 2-input multiplexer whose inputs come from two other multiplexers driven by the same control signal is fused to a 4-input multiplexer. This allows the efficient use of 6-input LUTs on current FPGAs.
- ROMs containing the same values are paired.

Additionally, in this step the design is pipelined and synchronized. In particular, if a control signal needs to be pipelined, the corresponding counters/timers are triggered in advance if possible. Otherwise, a reset value is computed for the registers that were added. As the design contains a loop, it must also be ensured that the head of a dataset does not collide with its tail anywhere. Therefore, the design is first generated in a sandbox to measure the latency of its different parts. In a second pass, the latency of the inner temporal permutation is then increased if needed. Conversely, if the latency of the inner part of a loop is higher than the duration of the dataset, it means that the amount of time (the gap) between two datasets must be increased. This information is used in the second pass for the temporal permutation to reduce as much as possible the number of elements of (U_i) (see Section 3.1) stored in ROM, while ensuring conflict-free addressing in the RAM.

Once these simplifications have been performed, the design is output as Verilog code.

Limitations. A limitation of our generator concerns the twiddle factors. In the designs we produce, each complex multiplier has a corresponding ROM that contains all the (real and imaginary) coefficients that it uses. A more distributed approach, along with a simple online computation of these coefficients could reduce further the number of BRAMs used.

3.3 RESULTS

In this section, we compare the cost and the performance of our generated FFT datapaths with other, state-of-the-art memory-efficient FFT architectures.

Table 3 lists the benchmarks we compare against. We consider two types of designs. The first type (A–D) is the prior iterative reuse structure from [47] exemplified in Fig. 2d, with different solutions for the streaming permutations. The original [47] uses the permutations from [63], which is A in the table. B and C use different solutions that are not specific to linear permutations. Both, A and B are available online at [44]. D improves the permutations in [63].

The second type (E–G) is the proposed architecture exemplified in Fig. 10b, again with different solutions for the necessary fused permutation block. E and F is what can be built with prior work that provides a general streamed permutation network. G is our proposed solution specialized to the two permutations that need to be fused. Note that neither E or F has been used within an FFT architecture as proposed here.

Table 3 analyzes the cost and performance for a radix-2 Pease FFT. We discuss these next.

Cost. For the memory consumption, we list the RAM requirement for the permutation part, excluding the memory used to store the twiddle factors. C theoretically should allow the use of banks of 2^{t-1} words for the perfect shuffle, but when used with the structure in Fig. 2d, the latency of the inner loop had to be increased to avoid dataset collisions, thus requiring 2^t words for all RAM banks. The gains compared to A–E are a factor of two or four; the only competitive method is F. However, the routing cost is at least a factor of two higher, and even more for t smaller than k.

For the routing requirements, we assume that the methods B, C, E, F using complete permutation networks implement them with [84], i.e., using $(k-1/2)\cdot 2^k$ 2×2-switches. We counted 2 multiplexers per switch, and 2^k multiplexers for the loop. Figure 15 plots the formulas in Table 3 for three different numbers of ports and a range of FFT sizes. Our method is better compared to A–D and F. Only E use less multiplexers[4] for large values of t, but requires four times more RAM banks.

In summary, we improve routing cost compared to F (and B and C) and RAM cost compared to A–E, both by at least a factor of two.

Gap. Next we analyze the minimal number of cycles between two datasets, i.e., the gap, which is the inverse of the throughput. In our case, it is constrained by the duration of the input itself (2^t cycles), and by the time the dataset stays in the loop. In Table 3, we assumed that the designs were all targeting $\approx$ 400Mhz on a Virtex 7. This requires a 4 cycles pipelining for the arithmetic part (butterfly and multiplications), and one register every 2 multiplexers (a complete permutation network has therefore a latency of $\lceil k/2 \rceil + 1$ cycles). Additionally, we assumed that all temporal permutations (even when fused) were done using the minimal possible latency; a feature that can

4 Using the RAM/SNW/RAM architecture instead would have yield better routing complexity, but for twice the amount of RAM.

Ref	Architecture	Permutation	Memory	Routing (2 : 1 Mux)	Gap (cycles per transform)
A	Fig. 2d [47]	Püschel [63]	2^{k+1} banks of 2^{t+1} words	$(\min(t,k)+3/2)\cdot 2^{k+1}$	$2^t+(n-1)\cdot\max(2^t,2^{t-1}+9)$
B	Fig. 2d [47]	Milder [45]	2^{k+2} banks of 2^{t+1} words	$(k-1/4)\cdot 2^{k+2}$	$\geqslant 2^t+(n-1)\cdot\max(2^t,2^{t-1}+\lceil k/2\rceil+8)$
C	Fig. 2d [47]	Koehn [38]	2^{k+1} banks of 2^t words	$(k-3/8)\cdot 2^{k+3}$	$2^t+(n-1)\cdot\max(2^t,2^{t-1}+2\lceil k/2\rceil+9)$
D	Fig. 2d [47]	SRS (Sec. 2.2.4)	2^{k+1} banks of 2^t words	$(\min(t,k)+3/2)\cdot 2^{k+1}$	$2^t+(n-1)\cdot\max(2^t,2^{t-1}+9)$
E	Fig. 10b (novel)	Milder [45]	2^{k+1} banks of 2^{t+1} words	$k\cdot 2^{k+1}$	$\geqslant 2^t+n\cdot\max(2^t,2^{t-1}+\lceil k/2\rceil+8)$
F	Fig. 10b (novel)	Koehn [38]	2^k banks of 2^t words	$(k-1/4)\cdot 2^{k+2}$	$2^t+n\cdot\max(2^t,2^{t-1}+2\lceil k/2\rceil+9)$
G	Fig. 10b (novel)	Proposed	2^k banks of 2^t words	$(\min(t,k)+1)\cdot 2^{k+1}-2$	$2^t+n\cdot\max(2^t,2^{t-1}+\lceil\min(t,k)/2\rceil+8)$

Table 3: Comparison of different architectures using different permutation methods, for a radix-2 Pease FFT, for $k>1$.

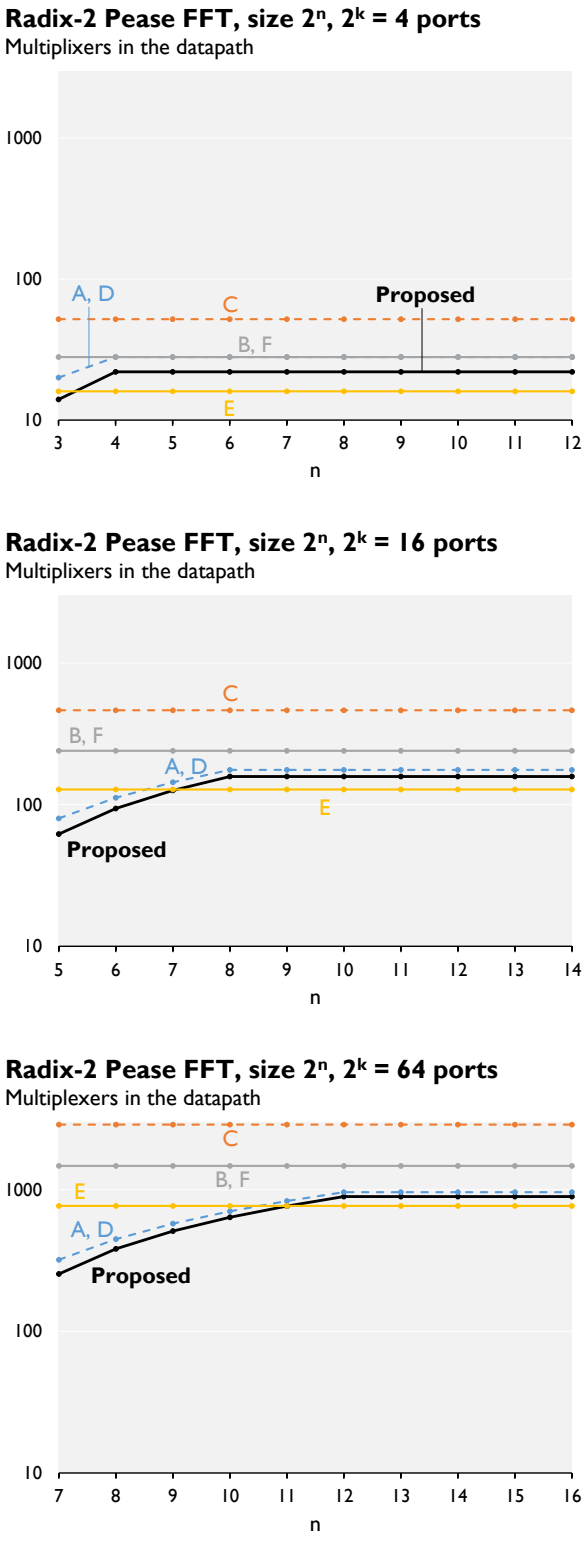


Figure 15: Number of 2-input multiplexers in the datapath of a radix-2 Pease FFT, for different streaming widths. Lower is better.

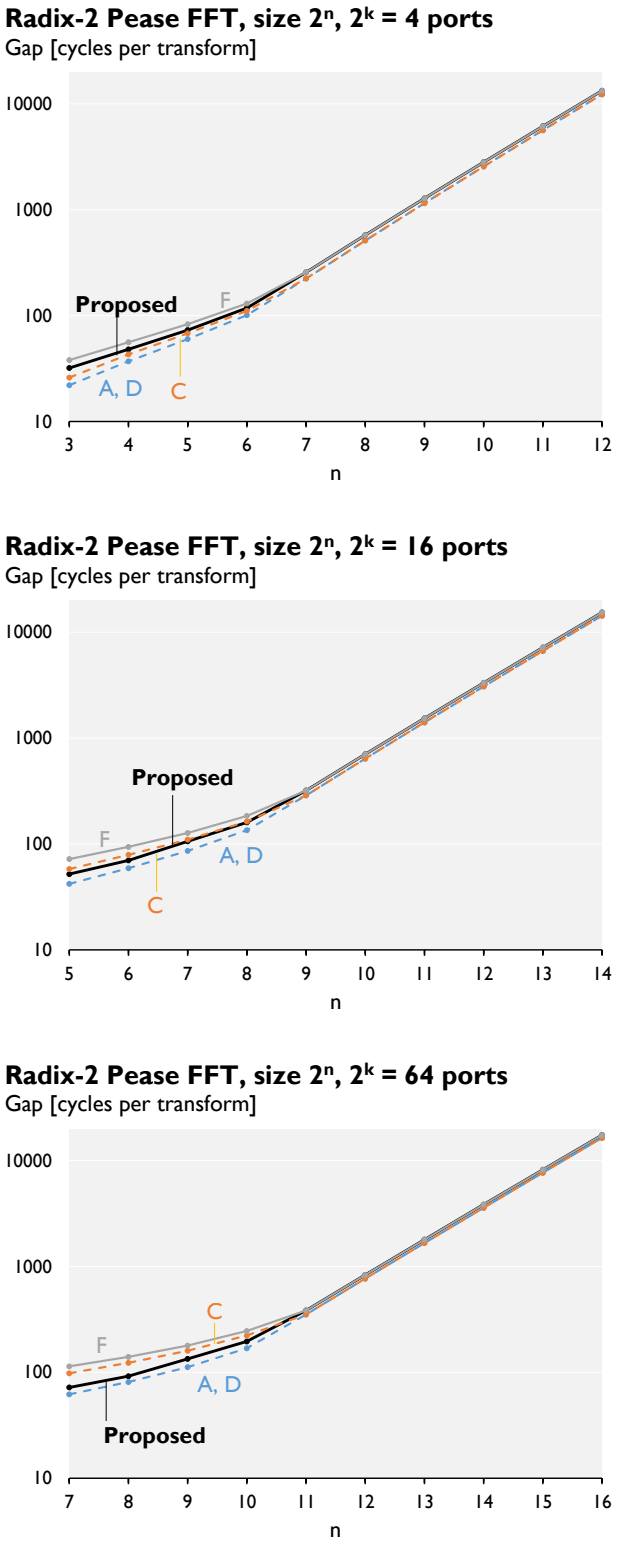


Figure 16: Gap of a radix-2 Pease FFT in number of cycles between two transforms, for different streaming widths. Lower is better.

easily be obtained using dual-ported RAM. However, with [45] (B and E), the total "temporal latency" depends on the chosen decomposition, and we can therefore only provide a lower bound. The corresponding formulas in Table 3 are plotted in Fig 16. It appears that, for $n - k = t \geqslant 5$, the latency required to avoid two datasets overlapping in the loop dominates the intrinsic inner latency of the loop. Thus, the term 2^t becomes the dominant term in the max, and the gap becomes $n \cdot 2^t$ for all streaming iterative reuse architectures (A–D), and $(n+1) \cdot 2^t$ for the fused permutation structure (E–G). Thus, as n, and hence t, increases the gaps of the different solutions converge. The same is then also true for the throughputs.

Results after place and route. Among the prior FFT solutions only A ([47]) is available online at [44]. We compare these designs with our solution G after place-and-route. For completeness we also implemented a variant of our generator [66] that produces the FFTs in D, which reduces the RAM cost of A.

The area and the RAM consumption of different designs for a radix-2 FFT are shown in Fig. 17, after place and route on a Xilinx Virtex 7 xc7vx1140 using Vivado 2014.4, using an element size of 16 bits. We observe that, as the number of RAM bank does not depend on n in the considered designs, the memory consumption stays constant until the capacity of the BRAM is reached. As expected from Table 3 our design requires fewer BRAMs; since the twiddle factors are stored in BRAMs as well, the RAM usage is not exactly halved. The logic area is roughly comparable and includes the control, which was not included in Table 3. While our control logic is arguably more efficient than storing all the switch configuration and addresses for all cycles, it is more complex than the one used in A or D, which explains why we do not require fewer slices.

Fig. 18 shows some results for a radix-4 FFT, which slightly reduces the number of multiplications needed but can only be folded at the granularity of DFT_4 blocks, which themselves are implemented using four butterflies. The overall behavior and comparison is analogous to the radix-2 case.

Discussion. Because our main target is FPGA, which contains BRAM modules, we compared our work with other RAM-based permutation techniques. However, other approaches based on registers [59] or distributed buffers [31] could be beneficial on platforms where grouping several memory elements does not improve the cost (ASICs).

Hardware architectures to compute DFTs are a classic topic in the literature, and other approaches that also use a RAM capacity equal to the size of the dataset (2^n) exist. However, these works are based on in-place algorithms [32], or consist of parameterized architectures [25] that do not provide the same flexibility as a generated streamed architecture (for instance, the number of ports is constrained by the radix used in the algorithm).

3.4 CONCLUSION

We proposed a novel method to design a datapath capable of realizing a number of fixed streamed linear permutations. As an application, we proposed a new variant of a folded Pease FFT that requires only one permutation block for both, the internal shuffles and the final bit reversal. While in some FFT applications, the bit reversal can be omitted, in many others it cannot, e.g., if frequency components need to be processed in order from low to high. For those, our new architecture offers novel Pareto-optimal tradeoffs between performance and logic/memory cost across an entire design space

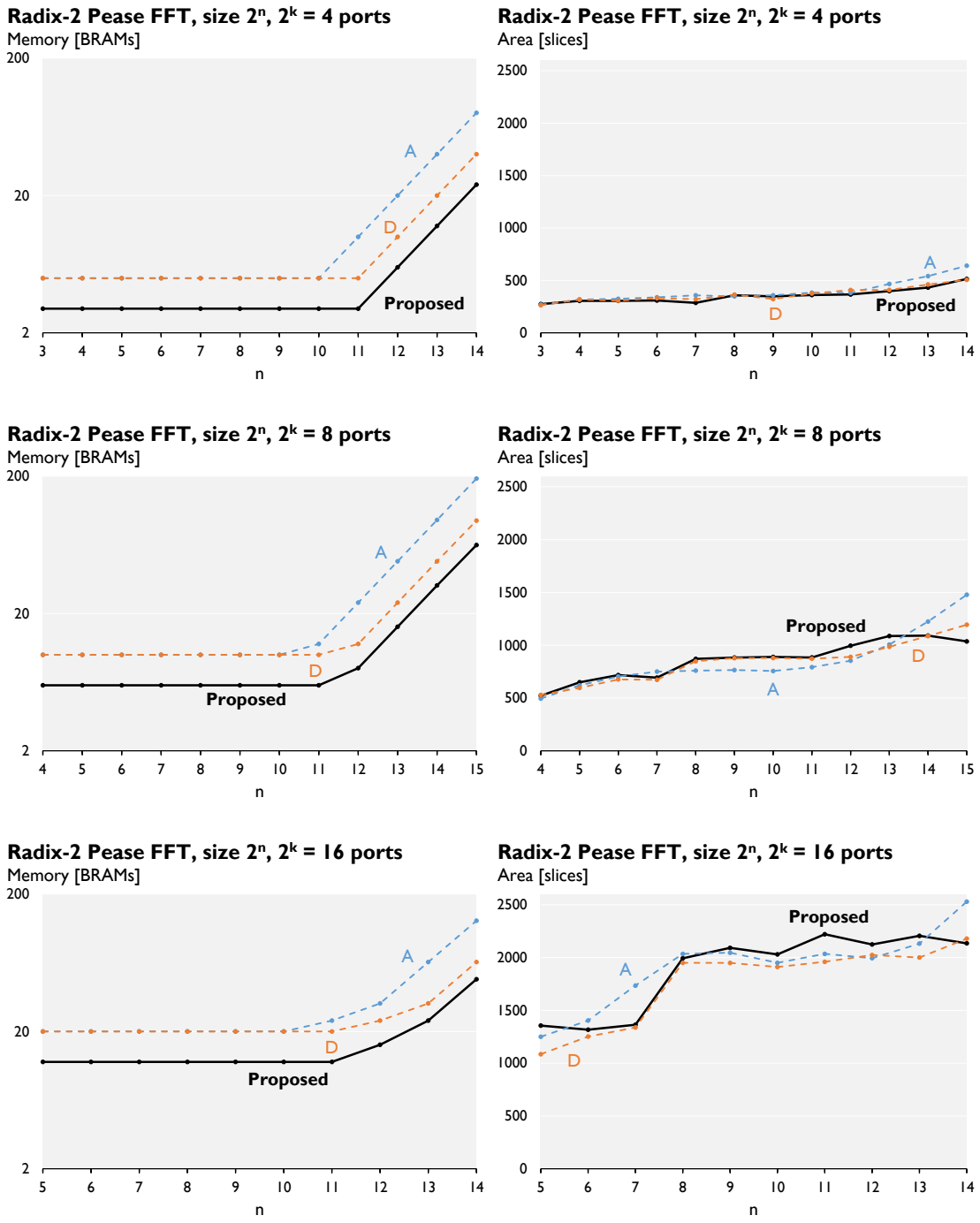


Figure 17: Resources used by a radix-2 Pease FFT. Lower is better.

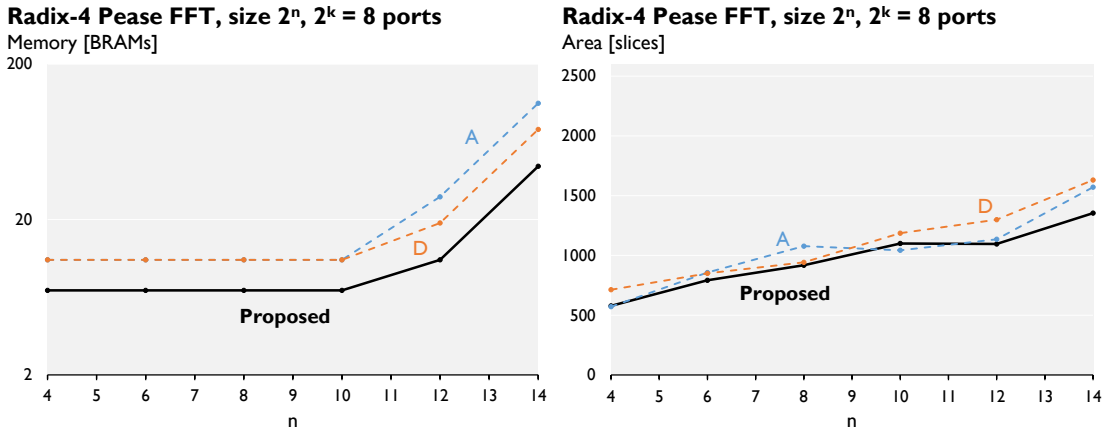


Figure 18: Resources used by a radix-4 Pease FFT. Lower is better.

of FFTs given by the chosen radix and number of input ports. These should directly translate to increased energy efficiency for a wide range of resource-constrained embedded applications.

4

IN SEARCH OF THE OPTIMAL WALSH-HADAMARD TRANSFORM FOR STREAMED PARALLEL PROCESSING

The *Walsh-Hadamard transform* (WHT) is an important function in signal processing [29, 5] and coding theory [41, 86]. Similar to the FFT, it is computed using a network of $n \cdot 2^{n-1}$ butterflies but without the twiddle factors in between (see Fig 19a for $n = 4$). The network can be modified in exponentially many ways by properly changing the permutations between stages (e.g., [35]). For example, Fig 19b shows a Pease-like WHT [60] that consists of equal stages suitable for iterative implementation in hardware.

Knowing the exact space of valid WHT networks makes it possible to search for the optimal one for a given implementation task. In this chapter, published in [75], we consider streaming implementations of the WHT (see Fig. 20), and ask the following question: For given n and k, which is the optimal WHT network for a streaming implementation, in terms of number of RAMs and switches required for implementing the corresponding streaming permutations. Towards answering this question we offer the following contributions:

- We exactly characterize the (exponentially large) set of all valid WHT networks such that the occurring permutations are linear.
- We present an algorithm to smartly search the large space of WHT networks at least for small sizes n.
- Using the search we find, for given n and k, novel and non-obvious WHT networks that have proven optimal hardware cost. An example is shown in Fig. 19c.
- Our results show the trend in hardware cost and give evidence that there are for all sizes n yet undiscovered WHT networks that are optimal for streaming.

Related work. The work in [35, 34] is similar in concept for WHT software implementations. It explores a large set of WHT algorithms to obtain efficient software library implementations, or as a test case for a model predicting the performances of libraries. However, the goal is to find the recursion best matched to the memory hierarchy; streaming in hardware poses a very different structural requirement.

4.1 BACKGROUND AND NOTATIONS

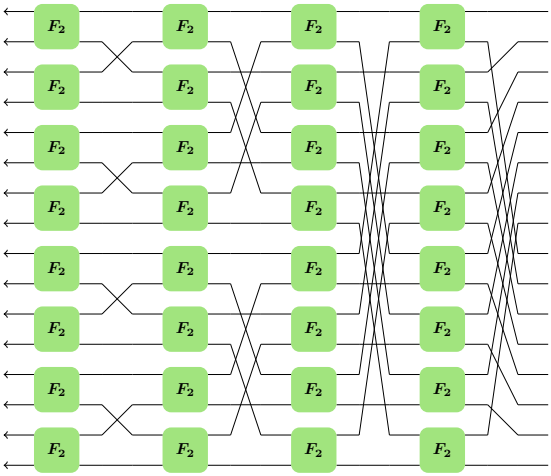
We provide background on Walsh-Hadamard transform (WHT) algorithms and their implementation in streaming hardware. Particularly important are the permutations between butterfly stages for which there is a large degree of freedom that we use as a search space to find the optimum.

WHT algorithms. The WHT is a linear transform that computes $y = H_{2^n} \cdot x$ where $x, y \in \mathbb{C}^{2^n}$. It is defined by the *Hadamard matrix*:

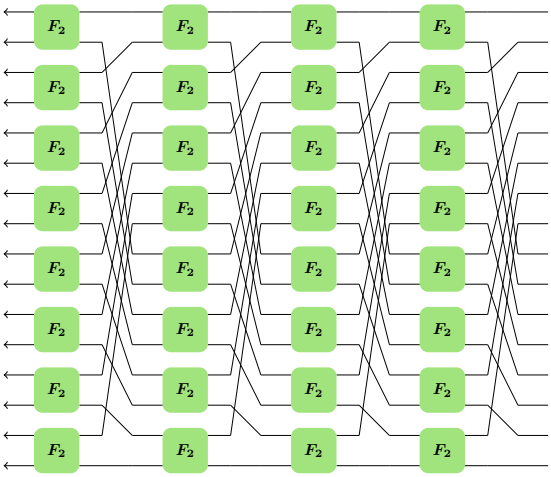
$$H_2 = \mathrm{DFT}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \text{ and } H_{2^n} = H_2 \otimes H_{2^{n-1}}, \text{ for } n > 1. \tag{24}$$

A direct computation shows that the Hadamard matrix can be rewritten as

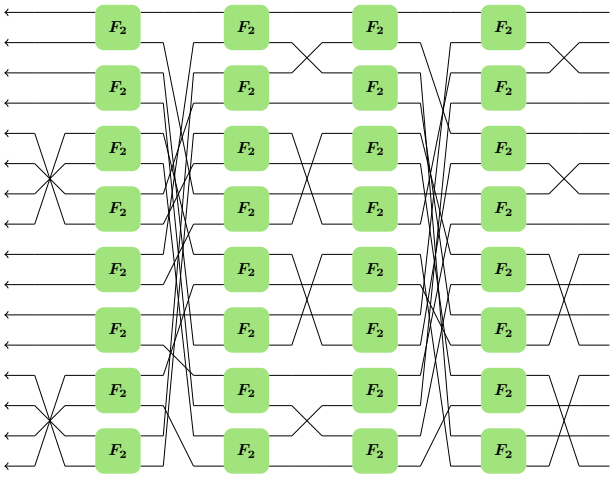
$$H_{2^n} = \left((-1)^{i_b^T j_b} \right)_{0 \leqslant i,j < 2^n}, \tag{25}$$



(a) Radix-$2^r = 2$ Iterative WHT (26)



(b) Radix-$2^r = 2$ Pease WHT (27)



(c) Algorithm that, once streamed with $2^k = 4$, yields an implementation that minimizes both the number of RAM stages, and the number of switch stages.

Figure 19: Dataflows computing a WHT on $2^n = 16$ elements. The H_2 blocks represent butterflies.

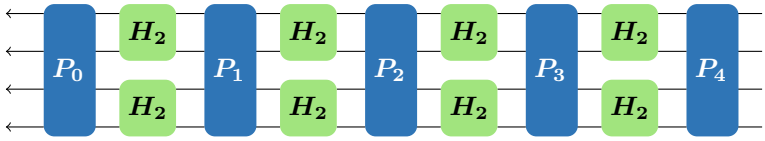


Figure 20: Algorithms from Fig. 19 folded with a streaming width $2^k = 4$. This architecture uses a fourth of the number of butterflies, but processes the inputs over $2^{n-k} = 4$ cycles.

using our previous notation that, for an integer $0 \leqslant i < 2^n$, denotes with i_b the n-bit column vector containing the binary representation of i with the most significant bit at the top.

For instance, the WHT on 8 elements corresponds to the matrix

$$H_{2^3} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{pmatrix}.$$

The definition, recursively applied, directly yields the algorithm, i.e., butterfly network, shown in Fig. 19a. It consists of n stages of 2^{n-1} butterflies $I_{2^{n-1}} \otimes H_2$. It is, in essence, a Cooley-Tukey FFT [15] without twiddle factors, requiring only $n \cdot 2^n$ operations. As for the FFT, the occuring permutations are all BPs; thus, the algorithm can be formally written using the Kronecker formalism [33, 82] as

$$H_{2^n} = \pi(P_0) \cdot \prod_{\ell=1}^{n} ((I_{2^{n-1}} \otimes H_2) \cdot \pi(P_\ell)), \tag{26}$$

where $P_0, \ldots, P_n$ are $n \times n$ permutation matrices. A given butterfly network, i.e., algorithm, can be manipulated in a myriad of ways to obtain variants, for example, by permuting the butterflies within stages. A popular example is the constant-geometry Pease-like [60] algorithm that has the same perfect shuffle $\pi(C_n)$ after every stage (see Fig. 19b). It is formally written as

$$H_{2^n} = \pi(I_n) \cdot \prod_{\ell=1}^{n} ((I_{2^{n-1}} \otimes H_2) \cdot \pi(C_n)). \tag{27}$$

We will later determine the exact set of possible linear permutations between stages to exhaustively search for the optimal WHT algorithm when used for the streaming implementations explained next.

Streaming WHT. A given WHT algorithm can be directly mapped to hardware but incurs $O(n \cdot 2^n)$ area cost. As with the FFT (Fig 2a), it is possible to reduce this cost to $O(n2^k)$ while maintaining high throughput by folding the network as shown in Fig. 20 [50]. For the implementation of the resulting SLPs, we use the SNW/RAM/SNW architecture (Section 2.2.4), as we consider RAMs more costly than the required switches.

4.2 ENUMERATION OF WHT ALGORITHMS

Besides the iterative and Pease WHT (respectively in Fig. 19a and Fig 19b), many other variants can be derived, corresponding to different butterfly networks. For example, the butterflies in one stage can be permuted with a permutation σ. Formally, this means replacing

$$I_{2^{n-1}} \otimes H_2 = (\sigma \otimes I_2) \cdot (I_{2^{n-1}} \otimes H_2) \cdot (\sigma^{-1} \otimes I_2). \tag{28}$$

We are only interested in the cases where $\sigma \otimes I_2$ is linear, which is true if and only if σ is linear, i.e., $\sigma = \pi(P)$, where P is an invertible $(n-1) \times (n-1)$ bit matrix ($P \in \mathrm{GL}_{n-1}(\mathbb{F}_2)$). Using $\pi(P) \otimes I_2 = \pi(P \oplus I_1)$, (28) becomes

$$I_{2^{n-1}} \otimes H_2 = \pi(P \oplus I_1) \cdot (I_{2^{n-1}} \otimes H_2) \cdot \pi(P^{-1} \oplus I_1). \tag{29}$$

Using these degrees of freedom, (27) was derived from (26) and one could wonder whether there are more valid transformation of the WHT algorithms.

The following theorem, proven in Chapter 7, is a main contribution of this thesis. It enables the enumeration of all linear permutations between stages that produce a valid WHT:

Theorem 3. *The Hadamard matrix H_{2^n} satisfies*

$$H_{2^n} = \pi(P_0) \cdot \prod_{\ell=1}^{n} ((I_{2^{n-1}} \otimes H_2) \cdot \pi(P_\ell)) \tag{30}$$

if and only if there exist $B \in \mathrm{GL}_n(\mathbb{F}_2)$ and $(Q_1, \cdots, Q_n) \in (\mathrm{GL}_{n-1}(\mathbb{F}_2))^n$ such that

$$\begin{aligned} P_0 &= B \cdot \begin{pmatrix} Q_1 & \\ & 1 \end{pmatrix}, P_n = \begin{pmatrix} & Q_n^{-1} \\ 1 & \end{pmatrix} \cdot B^T, \textit{ and} \\ P_\ell &= \begin{pmatrix} & Q_\ell^{-1} \\ 1 & \end{pmatrix} \cdot \begin{pmatrix} Q_{\ell+1} & \\ & 1 \end{pmatrix}, \textit{ for } 0 < \ell < n. \end{aligned} \tag{31}$$

In particular, there are a total of $g_{n-1}^n g_n$ butterfly networks, where $g_n = |\mathrm{GL}_n(\mathbb{F}_2)| = \prod_{i=0}^{n-1}(2^n - 2^i)$.

As an example, (27) corresponds to $Q_i = I_{n-1}$ for $1 \leqslant i \leqslant n$, and $B = I_n$. All networks for $n = 2$ are shown in Fig. 21 with associated bit matrices in Table 4.

Note that the theorem shows that in addition to (29), there is another, non-obvious degree of freedom: for any LP $\pi(B)$, the WHT satisfies

$$H_{2^n} = \pi(B) \cdot H_{2^n} \cdot \pi(B^T). \tag{32}$$

This degree of freedom will later produce novel optimal WHT algorithms for streaming implementations. Note that, in general, $\pi(B^T) \neq \pi(B)^T$.

4.3 SEARCH FOR OPTIMAL ALGORITHMS

The number of WHT algorithms $g_{n-1}^n g_n$ grows exponentially (see Table 5), which makes enumeration infeasible, except for very small sizes. Here we propose a search algorithms that pushes feasibility into the region of $n = 5$–8 to find evidence for the existence of better, yet unknown algorithms for streaming.

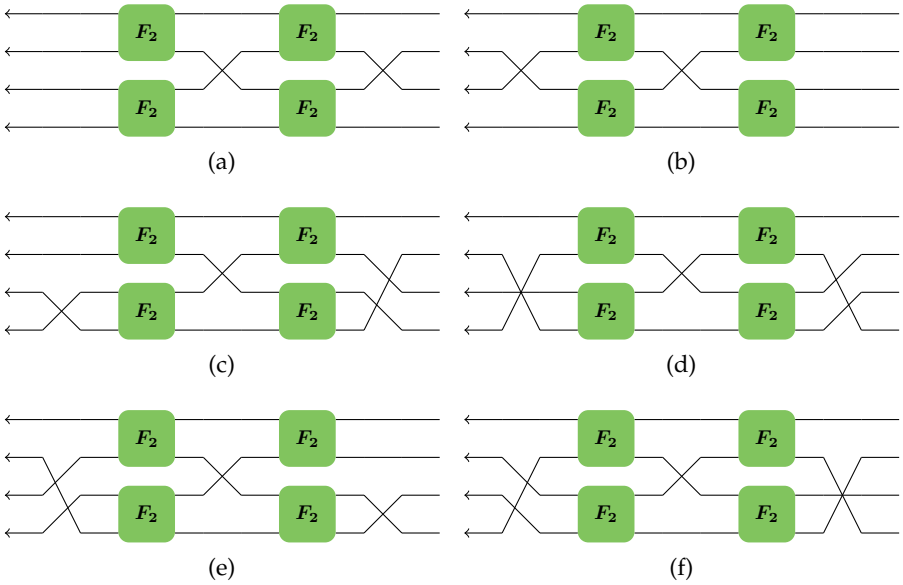


Figure 21: Dataflow of all butterfly networks with linear permutations computing H_{2^2}. The associated bit matrices are listed in Table 4

Ref.	P_0	P_1	P_2	$P_{0:n}$	X
(a)	$\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}$
(b)	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$
(c)	$\begin{pmatrix} 1 & \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & \\ 1 & 1 \end{pmatrix}$
(d)	$\begin{pmatrix} 1 & 1 \\ & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ & 1 \end{pmatrix}$
(e)	$\begin{pmatrix} 1 & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & \end{pmatrix}$
(f)	$\begin{pmatrix} & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & \end{pmatrix}$	$\begin{pmatrix} & 1 \\ 1 & 1 \end{pmatrix}$

Table 4: Matrices P_0, P_1 and P_2 of all butterfly networks with linear permutations computing H_{2^2}. The corresponding product of matrices ($P_{0:n}$) and spreading matrix (X) defined in Chapter 7 are presented as well. The first line corresponds to the Pease algorithm, the second one to its transpose. These two algorithms are the only ones that can be obtained using (24) recursively [35].

n	1	2	3	4	5	6	7	8
Derived from (24)	1	2	6	24	112	568	3032	16768
BP	1	2	48	31104	$\approx 10^9$	$\approx 2 \cdot 10^{15}$	$\approx 5 \cdot 10^{23}$	$\approx 2 \cdot 10^{34}$
Linear	1	6	18144	$\approx 4 \cdot 10^{12}$	$\approx 4 \cdot 10^{27}$	$\approx 10^{51}$	$\approx 7 \cdot 10^{84}$	$\approx 4 \cdot 10^{130}$

Table 5: Number of butterfly networks with linear permutations, number of butterfly networks with bit permutations and number of such networks that can be derived from (24) [35] for a given n.

4.3.1 *Cost function*

We search for WHT algorithms that minimize a given implementation cost. In this chapter, we minimize first the number of independent memory banks and then the number of multiplexers used to implement the corresponding streaming permutations with a streaming width of 2^k. Therefore, we use the SNW/RAM/SNW architecture (Section 2.2.4) that guaranties a minimal number of banks for a given SLP. Therefore, our cost function favors spatial permutations (i.e. matrices of the form (12)), and then minimize (17). First, we block each P_i as

$$P_i = \begin{pmatrix} P_{i,a} & P_{i,b} \\ P_{i,c} & P_{i,d} \end{pmatrix}, \text{ with } P_{i,d} \text{ of size } k \times k.$$

The cost function we minimize is

$$\text{Cost} = \sum_{i=0}^{n} \text{Cost}(P_i), \tag{33}$$

where

$$\text{Cost}(P_i) = n^2 \Delta_{P_i} + \max(\operatorname{rank} P_{i,c}, n - \operatorname{rank} P_{i,a} - \operatorname{rank} P_{i,d}).$$

Here, Δ_{P_i} denotes whether the SLP $(\pi(P_i), 2^k)$ is spatial; its value is 0 in this case (if $P_a = I_{n-k}$ and $P_b = 0$; no RAM stage is needed), and 1 otherwise (thus 2^k RAM banks are needed). The factor n^2 ensures that a RAM stage is more penalised than any possible number of switches. Further, it allows to retrieve the number of RAM stages needed via an euclidean division.

Any other *extensive* cost (i.e., that has a cost in the form (33), with $\text{Cost}(P_i) \geqslant 0$) can be used with the method we propose next. For instance, latency and total memory could as well be minimized.

4.3.2 *Search algorithm*

We assume a function G_ℓ that can enumerate all invertible $\ell \times \ell$ bit matrices; $G_\ell(i)$ is the i^{th} such matrix. Our approach consists of two main steps. We first compute a matrix $C^{\text{int}} = (c^{\text{int}}_{i,j})_{0 \leqslant i,j < g_{n-1}}$, containing the minimal cost of the internal permutations $\text{Cost}(P_1) + \cdots + \text{Cost}(P_{n-1})$ for each possible pair of matrices $Q_1 = G_{n-1}(i)$ and $Q_n = G_{n-1}(j)$. Then, we compute similarly a matrix C^{ext} containing the minimal cost of the external permutations $\text{Cost}(P_0) + \text{Cost}(P_n)$. The optimal cost is then the smallest element of $C^{\text{int}} + C^{\text{ext}}$.

Internal cost matrix C^{int}. We first store in a matrix $C = (c_{i,j})_{0 \leqslant i,j < g_{n-1}}$ the cost that a single internal permutation $\pi(P_\ell)$ would have, given $Q_\ell = G_{n-1}(i)$ and $Q_{\ell+1} = G_{n-1}(j)$:

$$c_{i,j} = \text{Cost}\left(\begin{pmatrix} & G^{-1}_{n-1}(i) \\ 1 & \end{pmatrix} \cdot \begin{pmatrix} G_{n-1}(j) & \\ & 1 \end{pmatrix} \right).$$

The minimal cost that two consecutive internal permutations $\pi(P_\ell)$ and $\pi(P_{\ell+1})$ would have, given $Q_\ell = G_{n-1}(i)$ and $Q_{\ell+2} = G_{n-1}(j)$ is

$$\min_{Q_{\ell+1}} \text{Cost}(P_\ell) + \text{Cost}(P_{\ell+1}) = \min_k c_{i,k} + c_{k,j}.$$

This computation corresponds to the distance product [20] of C with itself. Getting C^{int} consists of performing this task $n-1$ times: $C^{\text{int}} = C^{n-1}$. We use a fast exponentiation algorithm, leading to an arithmetic complexity in $O(g_{n-1}^3 \log(n))$, and a memory footprint in $O(g_{n-1}^2)$ for this step.

External cost matrix C^{ext}. The matrix C^{ext} of the cost of the external permutations can be obtained by trying all the possible matrices for B, for each pair Q_1, Q_n. This step has an arithmetic complexity in $O(g_n g_{n-1}^2)$.

This search can be simplified for a given cost. For instance, for the cost shown in the result section, a close formulation for an optimal B was possible, making the second step negligible compared to the first. Further, our algorithm can be restricted to a subset of all linear permutations as shown in the results.

4.4 RESULTS

We implemented the search algorithm using as building block a specially designed linear algebra library for the efficient computation with 8×8 bit-matrices. Our cost function first minimizes the number of stages of RAM banks (i.e., favoring spatial SLPs), and then the number of switching stages (i.e., favoring small values for $\max(\operatorname{rank} P_2, n - \operatorname{rank} P_1 - \operatorname{rank} P_4)$). However, the arithmetic complexity of the algorithm prevented us from completing the search for $n \geqslant 6$ (which would have taken years). Therefore, we have also run the search restricted to permutations that are BPs (thus reducing the complexity to $O((n-1)!^3 \log(n))$). This choice is also theoretically important as all-known WHT networks (just as power-of-two FFT networks) are built using BPs. We will see that BPs alone will not yield the optimum.

Number of RAM stages. The Pease algorithm, when streamed, requires n stages of RAM banks, the iterative WHT only $n-k+1$. The minimal number of RAM stages found by our search for both LPs and also when restricted to BPs is $\lceil n/k \rceil$ and thus better. Note that if k divides n this cost can be achieved using an iterative radix-2^k algorithm. For k not dividing n our search finds novel solutions.

Number of switching stages. The minimal number of switching stages over all RAM-optimal algorithms found by our search is shown in Table 6. Note that in some cases, the best algorithm that uses BPs has more switches than the iterative algorithm. In these cases, the latter is not RAM-optimal.

Most interestingly, for streaming widths $2^k > 2$ our search with LPs discovers novel WHT networks that improve prior ones in both RAM usage and required switches. Considering BPs alone (as in all known network variants including the large space in [35]) is not sufficient. Further, all optimal networks found have a non-trivial B in (32).

Unfortunately, we did not manage to extrapolate from the WHT networks found to optimal solutions for all sizes n and k. To illustrate the difficulty of such an extrapolation, consider the optimal network found for $n = 4$ and $k = 2$ in Fig. 19c, and for $n = 5$ and $k = 3$ in Fig. 22. In addition, the linear permutations in these algorithms have a minimum number of bits set in their bit-matrices.

4.5 CONCLUSION AND FUTURE WORK

We introduced an idea that is of both theoretical and practical interest: namely, for an algorithm with regular structure and a given implementation task, one can enumerate all possible variants to find the optimal solution. The challenge is in characterizing

Log of size (n)	2	3		4			5				6					7					
Log of str. width (k)	1	1	2	1	2	3	1	2	3	4	1	2	3	4	5	1	2	3	4	5	6
Radix-2 Pease	4	6	6	8	8	8	10	10	10	10	12	12	12	12	12	14	14	14	14	14	14
Radix-2 Iterative	4	6	4	8	6	4	10	8	6	4	12	10	8	6	4	14	12	10	8	6	4
Best with BP	4	6	4	8	8	4	10	10	8	4	12	12	12	8	4	14	14	12	12	8	4
Best with LP	4	6	3	8	5	3	10	6	5	3	12	?	?	?	3	?	?	?	?	?	?

Table 6: Number of stages of 2^{k-1} switches in WHTs of size 2^n implemented with a streaming width of 2^k.

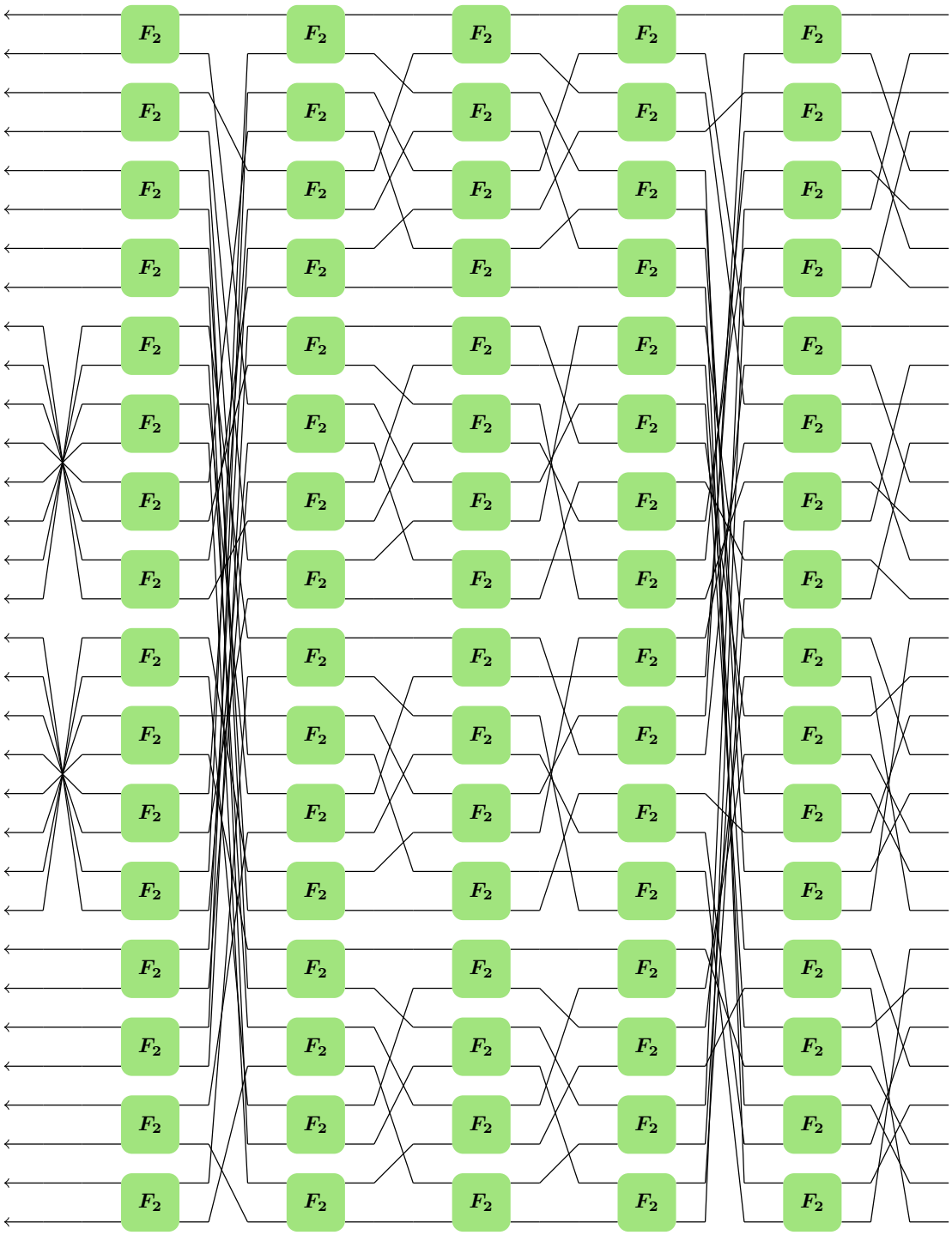


Figure 22: Butterfly network computing a WHT of size $2^n = 32$ that, when streamed with $2^k = 8$, yields an implementation with minimal number of memory elements and switches.

the space of algorithmic variants, which we did for the WHT, revealing the existence of algorithms that have strictly better RAM and logic requirements than what was previously possible.

We considered the WHT for its simplicity but the idea is in principle applicable to other regular algorithms. We give here a glance of the size of the space of sorting networks algorithms, and of FFT variants:

FFT. Twiddle factors make the characterization of FFT datapaths more complicated. Merging them with the butterflies yields promising results:

Proposition 9. *For a positive integer* n*, and* $n+1$ *invertible* $n \times n$ *bit matrices* $P_0, \dots, P_n \in \mathrm{GL}_n(\mathbb{F}_2)$*, the existence of* $n \cdot 2^{n+1}$ *complex numbers* $z_0^1, \dots, z_{2^{n+1}-1}^1, z_0^2, \dots, z_{2^{n+1}-1}^n$ *such that :*

$$\mathrm{DFT}_{2^n} = \pi(P_0) \cdot \prod_{k=1}^{n} \left(\left(\bigoplus_{\ell=0}^{2^{n-1}-1} \begin{pmatrix} z_{4\ell}^k & z_{4\ell+1}^k \\ z_{4\ell+3}^k & z_{4\ell+2}^k \end{pmatrix} \right) \cdot \pi(P_k) \right).$$

only depends on $P_{0:n}$ *and the spreading matrix* X (71).

A proof of this proposition is available upon request. Additionally, an exhaustive search shows that for $n \leqslant 4$, the set of pairs $(X, X^{-1}P_{0:n})$ such that there exist such complex numbers is a Cartesian product.

As an example, for $n = 4$, the valid FFT datapaths are those that satisfy:

$$X \text{ invertible}, X^{-1} = \begin{pmatrix} * & * & * & * \\ * & 1 & * & * \\ * & & 1 & * \\ * & & & 1 \end{pmatrix} \text{ and } X^{-1}P_{0:n} = \begin{pmatrix} & & * & * \\ & & * & * \\ & 1 & * & * \\ 1 & * & * & * \end{pmatrix},$$

where each symbol $*$ can be replaced by any value in $\mathbb{F}_2$.

Sorting networks. Sorting networks SN_{2^n} use more stages than WHTs or FFTs, which hinder the definition of a spreading matrix, and prevents an approach like the one we had for WHT butterfly networks. However, sorting networks present some degrees of freedom similar to the ones we found for the WHT. For all $B \in \mathrm{GL}_n(\mathbb{F}_2)$ and $Q \in \mathrm{GL}_{n-1}(\mathbb{F}_2)$,

$$\mathrm{SN}_{2^n} = \mathrm{SN}_{2^n} \cdot \pi(B), \quad \text{and} \tag{34}$$

$$I_{2^{n-1}} \otimes \mathrm{SN}_2 = \pi(Q \oplus I_1) \cdot (I_{2^{n-1}} \otimes \mathrm{SN}_2) \cdot \pi(Q^{-1} \oplus I_1). \tag{35}$$

The first degree of freedom (34) translates the fact that permuting the input of a sorting network doesn't change the result, and (35) means that we can swap SN_2 blocs like the butterflies in a WHT.

Additionally, for any vector $u \in \mathbb{F}_2^{n-1}$, the permutation $\pi \begin{pmatrix} I_{n-1} & \\ u^T & 1 \end{pmatrix}$ exchanges pairs of elements. Therefore, we have:

$$I_{2^{n-1}} \otimes \mathrm{SN}_2 = (I_{2^{n-1}} \otimes \mathrm{SN}_2) \cdot \pi \begin{pmatrix} I_{n-1} & \\ u^T & 1 \end{pmatrix}, \quad \text{and} \tag{36}$$

$$I_{2^{n-1}} \otimes \mathrm{SN}_2 = \pi \begin{pmatrix} I_{n-1} & \\ u^T & 1 \end{pmatrix} \cdot \left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^{u^T i_b} \right). \tag{37}$$

Algorithm considered	Number of RAM stages	Number of switches stages	Conjecture?
Radix-2 Pease	n	$2n$	No
Radix-2 iterative	$n-k+1$	$2(n-k+1)$	No
Best with BP	$\lceil n/k \rceil$	?	Yes, holds for $n \leqslant 8$
Best with LP	$\lceil n/k \rceil$	?	Yes, holds for $n \leqslant 5$

Combining (35), (36) and (37) shows that, for all function

$$f : \{0, \dots, 2^{n-1} - 1\} \to \mathbb{F}_2,$$

for all matrix $Q \in \mathrm{GL}_{n-1}(\mathbb{F}_2)$, and for all pair of vectors $u, v \in \mathbb{F}_2^{n-1}$, there exists a function

$$g : \{0, \dots, 2^{n-1} - 1\} \to \mathbb{F}_2$$

such that

$$\bigoplus_{i=0}^{2^{n-1}-1} X_2^{f(i)} = \pi \begin{pmatrix} Q & \\ u^T & 1 \end{pmatrix} \cdot \left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^{g(i)} \right) \cdot \pi \begin{pmatrix} Q^{-1} & \\ v^T & 1 \end{pmatrix}. \tag{38}$$

These degrees of freedom may not be the only ones, but this already shows that the design space for streaming networks is considerably larger than the one for WHTs. Fully characterizing it, and searching through it for an optimal streaming implementation is a challenging problem.

Part II

IMPLEMENTATION

5

A DSL-BASED HARDWARE GENERATOR IN SCALA FOR FAST FOURIER TRANSFORMS AND SORTING NETWORKS

In this chapter, published in [74] and that extends the work published in [72], we present a hardware generator for computations with regular structure including the fast Fourier transform (FFT), sorting networks, and others. The input of the generator is a high-level description of the algorithm; the output is a token-based, synchronized design in the form of RTL-Verilog. Building on prior work, the generator uses several layers of domain-specific languages (DSLs) to represent and optimize at different levels of abstraction to produce a RAM- and area-efficient hardware implementation. Two of these layers and DSLs are novel. The first one allows the use and domain-specific optimization of the streaming permutations and the algorithmic structures seen in the first part of this thesis. The second DSL enables the automatic pipelining of a streaming hardware dataflow and the synchronization of its data-independent control signals.

The generator including the DSLs are implemented in Scala, leveraging its type system, and uses concepts from lightweight modular staging (LMS) to handle the constraints of streaming hardware. Particularly, these concepts offer genericity over hardware number representation, including seamless switching between fixed-point arithmetic and FloPoCo generated IEEE floating-point operators, while ensuring type-safety. In addition, automatic simplifications that exploit the particular structure of the specified implementation are performed at each stage, thus reducing both ROM consumption and DSP slices, and improving performance in terms of latency and throughput.

We show benchmarks of generated FFTs, sorting networks and Walsh-Hadamard transforms that outperform prior generators.

5.1 GENERATION PIPELINE

We will refer to all functions that our generator implements as transforms. These include the discrete Fourier transform, the Walsh-Hadamard transform, and sorting networks. Our proposed generator receives as input the desired transform, its size (a power of two), and some parameters that control the design space (e.g., streaming width, iterative reuse applied or not, hardware arithmetic representation). The output is the corresponding design in the form of RTL Verilog. The generation process consists of the three layers pictured in Fig. 23. Each of these layers employs a DSL to represent, manipulate, and optimize the algorithm at different levels of abstraction. Each DSL is implemented as embedded DSL inside Scala, and staging is used to allow manipulation. We first give a brief overview and then discuss the last two layers in greater detail in subsequent sections.

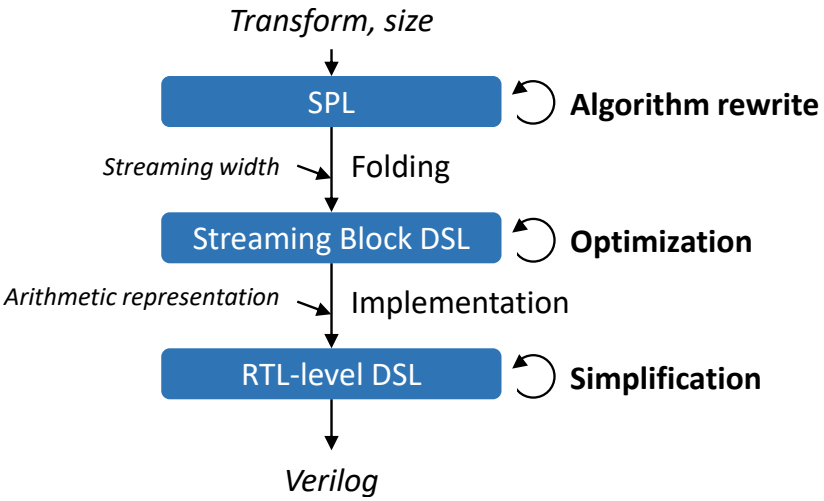


Figure 23: The different layers of our generator.

Operator	Description
First-order operator	
DFT_{2^n}	Discrete Fourier transform for input size 2^n
H_{2^n}	Walsh-Hadamard transform for input size 2^n
SN_{2^n}	Sorting network for 2^n inputs
$\pi(P)$	Linear permutation associated with the invertible bit-matrix P [68]
T_i, T_i'	Twiddle factors
X_2^c	Configurable two-input sorter
Higher-order operator	
$A \cdot B$	Composition of operators A and B
$\prod_i A_i$	Enumerated composition of operators A_i
$\bigoplus_i A_i$	Enumerated direct sum (parallel composition) of operators A_i

Table 7: SPL operators used in our generator.

5.1.1 *SPL*

The first step for generating a hardware implementation consists of choosing a suitable algorithm. Following [47], we represent these algorithms as *breakdown rules* that decompose a large transformation into smaller ones. These rules are represented using SPL, a mathematical language that represents linear algebra operations by matrices and operators on these matrices [64, 85, 33]. We introduce next SPL, and describe the rules we use for the different transforms we consider. Our implementation of this DSL in Scala is similar as in [54], and includes the operators in Table 7. Higher-order operators are used to recursively construct algorithms from first-order operators.

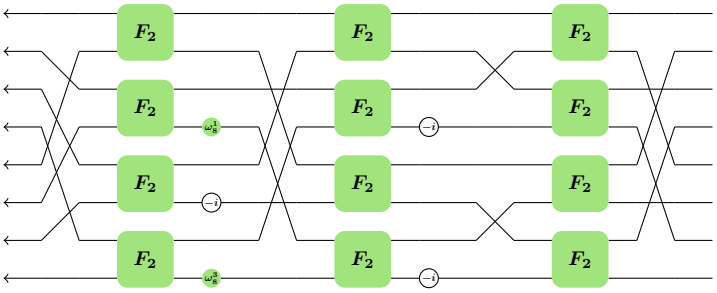


Figure 24: Radix-2 Cooley-Tukey FFT datapaths operating on $2^n = 8$ elements. This algorithm is used when iterative reuse is not enabled, as the permutations involved require less resources when streamed.

DFT. In the case where the desired transform is a DFT, our generator uses two breakdown rules. The first one, used in Fig. 2, is the constant-geometry radix-2^r Pease FFT [60]:

$$\mathrm{DFT}_{2^n} = \pi(J^r_n) \cdot \prod_{\ell=0}^{n/r-1} \left(T_{n/r-\ell-1} \cdot \left(\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r} \right) \cdot \pi(C^r_n) \right). \tag{39}$$

where T_ℓ is a diagonal matrix that performs element-wise complex multiplications with the twiddle-factors[1], $\pi(J^r_n)$ and $\pi(C^r_n)$ are permutations (respectively the radix-2^r-reversal, and the stride-by-2^r permutation), and $\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r}$ represents 2^{n-r} parallel DFTs of size 2^r each. As the expression within the product is always the same, this algorithm is well suited for iterative reuse. However, for designs with streaming reuse only, or for the generation of the base case 2^r-FFT, the radix-2^r Cooley-Tukey FFT (see Fig. 24) is used instead, as the permutations it requires use less resources.

$$\mathrm{DFT}_{2^n} = \pi(S^{n-r}_n) \cdot \left(\prod_{\ell=0}^{n/r-1} \left(\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r} \right) \cdot T'_\ell \cdot \pi(Q_\ell) \right) \cdot \pi(J^r_n), \tag{40}$$

where T'_ℓ is a diagonal matrix, and $\pi(Q_\ell)$ a permutation.

WHT. The algorithms we are using to compute a WHT are similar to the one used for DFTs, but do not include twiddle factors nor a final bit-reversal permutation. As an example, the Pease-like WHT algorithm is expressed as

$$H_{2^n} = \prod_{j=0}^{n-1} \left(\left(\bigoplus_{i=1}^{2^{n-1}} \mathrm{DFT}_2 \right) \cdot \pi(C_n) \right). \tag{41}$$

SN. Sorting networks (SNs) are somewhat similar to FFTs or WHTs but require a different form of butterflies, which are two-input sorters and thus nonlinear. Thus an extension to SPL is required as described in [89] following concepts from [22]. Formally, a two-input sorter is described as X^c_2. If $c = 0$, it sorts the two inputs in ascending order, if $c = 1$ it sorts them in descending order. With this, we can express

1 $T_\ell = \bigoplus_{i=0}^{2^{n-r}-1} \bigoplus_{j=0}^{2^r-1} \omega^{j \cdot i_{(r\ell)}}$, where ω is the principal 2^n-th root of unity, and $i_{(r\ell)}$ means that the $r\ell$ least significant bits of i have been set to 0.

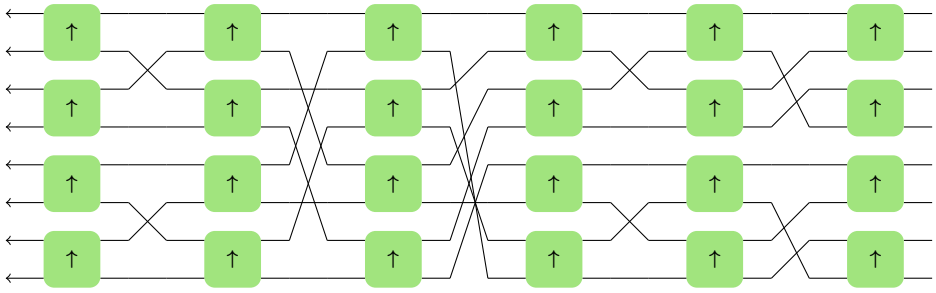


Figure 25: Batcher bitonic sorting network [4] operating on $2^n = 8$ elements. This design corresponds to the SN1 architecture in [89].

SNs using the previous formalism. For streaming reuse only, we use a Batcher bitonic SN [4] (see Fig. 25)::

$$\mathrm{SN}_{2^n} = \prod_{j=0}^{n-2} \left(\left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^0 \right) \cdot \prod_{\ell=0}^{n-i-2} \left(\pi(P_\ell) \cdot \left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^0 \right) \right) \cdot \pi(Q_j) \right) \cdot \bigoplus_{i=0}^{2^{n-1}-1} X_2^0. \tag{42}$$

This corresponds to the architecture SN1 in [89].

When iterative reuse is desired, a Pease-like network is used [79]:

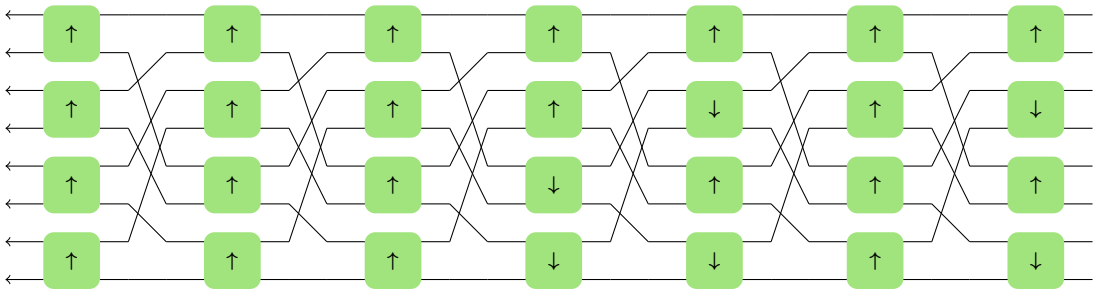
$$\mathrm{SN}_{2^n} = \left(\bigoplus_{i=1}^{2^{n-1}} X_2^0 \right) \cdot \prod_{j=0}^{n^2-n-3} \left(\pi(S_n) \cdot \left(\bigoplus_{i=1}^{2^{n-1}} X_2^{f(i,j)} \right) \right), \tag{43}$$

where f is a binary function. This second algorithm corresponds to the architecture SN5 of [89], with the following improvements:

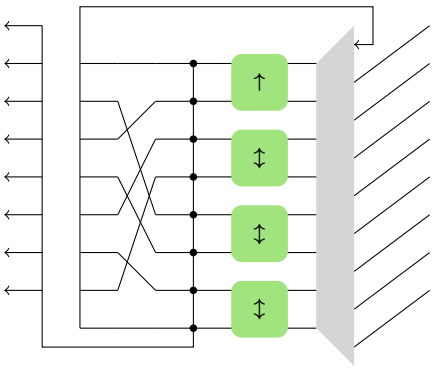
- The first $n-1$ stages are removed, as these correspond to a fixed permutation of the inputs, for which the order does not matter. This allows an increase of the throughput of the implementations.
- In [89], X_2^c had an additional pass-through configuration. We change the stages that used this mode such that the sorters perform useless comparisons instead (by copying the configuration of the stage located n places later). Therefore, we only use X_2^c as a sorter or as an inverted sorter, thus reducing the complexity of the implementation.
- When folded for iterative reuse, a loop with an early termination (similar to the structure used in Chapter 3 when fusing permutations) allows to simultaneously implement a single stage of sorters while performing only the necessary number of permutations (see Fig. 26b).

5.1.2 *Streaming-block DSL*

In the second step of the generator (Fig. 23), the SPL expression is formally folded according to the *streaming width*, i.e., the number of elements of the dataset that the design would be able to handle in each cycle. In the second step, the SPL expression

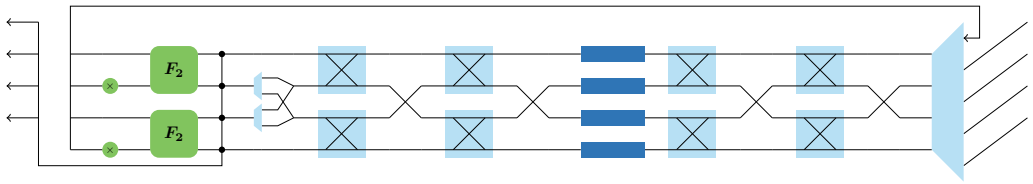


(a) Unfolded

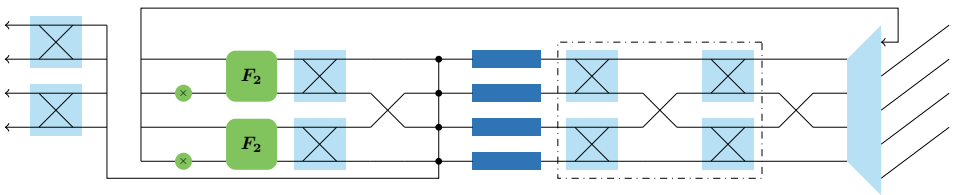


(b) Iterative reuse

Figure 26: Constant-geometry sorting network [79] operating on $2^n = 8$ elements. This design loosely corresponds to the SN5 architecture in [89].



(a) Before optimization.



(b) After optimization.

Figure 27: Design of Fig. 10b expressed using the streaming block DSL. The necessary streaming permutation is expanded into switches and RAM banks (dark blue rectangles). The optimization here "unrolls" some parts of the permutation to remove an array of multiplexer. Additionally, two arrays of switches were grouped for later mapping to 4-to-1 multiplexers. These optimizations increase the throughput and reduce the area of the final design.

Operator	Description	SPL correspondence
First-order operator		
$\bigoplus \mathrm{DFT}_2$	Butterfly array (add and substract its two inputs)	$\bigoplus \mathrm{DFT}_2$
T_i, T_i'	Twiddle factors	T_i, T_i'
X_2^c	Configurable two-input sorter	X_2^c
$\pi_i(P_0, \ldots, P_\ell)$	Array of multiplexers	$\pi \begin{pmatrix} I_t & \\ & P_i \end{pmatrix}$
$\sigma_i(\nu_0, \ldots, \nu_\ell)$	Single array of switches	$\pi \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ \nu_i^T & & & 1 \end{pmatrix}$
$\sigma_i'((u_0, \nu_0), \ldots, (u_\ell, \nu_\ell))$	Double array of switches	$\pi \begin{pmatrix} I_t & & & & \\ & & 1 & & \\ & & & \ddots & \\ \nu_i^T & & & & 1 \\ u_i^T & 1 & & & \end{pmatrix}$
$\tau_i((A_0, B_0), \ldots, (A_\ell, B_\ell))$	Array of RAM banks	$\pi \begin{pmatrix} A_i & B_i \\ & I_k \end{pmatrix}$
Higher-order operator		
$A_0 \cdot A_2 \cdots A_\ell$	Composition (without iterative reuse)	$\prod_{i=0}^{\ell} A_i$
$\prod_i A_i$	Composition with iterative reuse (A is implemented only once)	$\prod_{i=0}^{\ell} A_i$
$\prod_i^{\ell}(\overline{A_i} B_i)$	Composition with iterative reuse and early termination	$B_\ell \prod_{i=0}^{\ell-1}(A_i B_i)$

Table 8: Streaming blocks used in the streaming-block DSL. The last higher-order operator allows to represent the structure introduced in Chapter 3.

is formally folded according to the *streaming width*, i.e., the number of elements of the dataset that the design would be able to handle in each cycle. This includes inserting the necessary datapaths for the streaming permutations from Chapter 2. The DSL used thus expands SPL to include the streaming width (similar to the so-called Hardware-SPL in [47]), but also the following needed streaming blocks:

During this stage, a set of rewriting rules is used to simplify the streaming blocks, particularly in the case of a fused permutation. As an example, a radix-2 Pease DFT

on 8 elements (39), folded with a streaming width of 4 ports using iterative reuse with fused permutation would be represented in the streaming block DSL by

$$\prod_{i=0}^{3}\left(\overline{T_{2-i}\cdot\bigoplus_{\ell=1}^{4}\mathrm{DFT}_2\cdot\pi_i\left(I_2,S_2,S_2,S_2\right)}\cdot\right.$$
$$\sigma_i\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right)\cdot\pi(S_2)\cdot\sigma_i\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right)\cdot\pi(S_2)\cdot$$
$$\tau_i\left(\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0&1\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right)\right)\cdot\pi(S_2)\cdot$$
$$\left.\sigma_i\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right)\cdot\pi(S_2)\cdot\sigma_i\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right)\cdot\pi(S_2)\right),\quad(44)$$

which corresponds to the dataflow pictured in Fig. 27a.

In this expression, the array of multiplexers $\pi_i\left(I_2,S_2,S_2,S_2\right)$ does not perform anything during the last iteration. It is therefore interesting to "push" it after the early termination of the loop, as it can be implemented using only a rewiring $\pi(S_2)$. The array of switches that comes next, $\sigma_i\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right)$, does not permute anything during the first three iterations of the loop. It can be therefore "unrolled" into a non-parameterized array of switches $\sigma\left(\begin{pmatrix}1\\0\end{pmatrix}\right)$, thus saving logic and latency in the loop and therefore increasing throughput. At this point, the expression becomes

$$\sigma(\begin{pmatrix}1\\0\end{pmatrix})\cdot\prod_{i=0}^{3}\left(\overline{T_{2-i}\cdot\bigoplus_{\ell=1}^{4}\mathrm{DFT}_2\cdot\pi(S_2)}\cdot\pi(S_2)\cdot\sigma_i\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right)\cdot\pi(S_2)\cdot\right.$$
$$\tau_i\left(\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0&1\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right)\right)\cdot\pi(S_2)\cdot$$
$$\left.\sigma_i\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right)\cdot\pi(S_2)\cdot\sigma_i\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right)\cdot\pi(S_2)\right).$$

Continuing these optimizations, and regrouping the two rightmost single arrays of switches finally yields the expression

$$\sigma(\begin{pmatrix}1\\0\end{pmatrix})\cdot\prod_{i=0}^{3}\left(\overline{T_{2-i}\cdot\bigoplus_{\ell=1}^{4}\mathrm{DFT}_2\cdot\sigma(\begin{pmatrix}1\\0\end{pmatrix})\cdot\pi(S_2)}\cdot\right.$$
$$\tau_i\left(\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0&1\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right),\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1&0\end{pmatrix}\right)\right)\cdot\pi(S_2)\cdot$$
$$\left.\sigma'_i\left(\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right),\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right),\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right),\left(\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right)\right)\cdot\pi(S_2)\right),\quad(45)$$

pictured in Fig. 27b.

5.1.3 *Streaming-RTL DSL*

In the final stage, the streaming blocks are transformed into a dependency graph where each node, called a *signal*, represents a hardware operator that outputs one value per cycle. A signal may have zero (constant signals, inputs, timers and counters),

Operator	Description	Example
Constant signal		
`value` is a numerical value.		
`Const(value)`	Constant signal	`Const(5.3)`
Register signal		
`input` is a signal.		
`Register(input)`	Flip/flop register	`input.register()`
Arithmetic signals		
`lhs` and `rhs` are signals of the same type.		
`Plus(lhs, rhs)`	Sum of the operands	`lhs + rhs`
`Minus(lhs, rhs)`	Difference of the operands	`lhs - rhs`
`Times(lhs, rhs)`	Product of the operands	`lhs * rhs`
`And(lhs, rhs)`	Binary AND of the operands	`lhs & rhs`
`Xor(lhs, rhs)`	Binary XOR of the operands	`lhs ∧ rhs`
Memory signals		
`content` is an indexed sequence of numerical values,		
`address` is an unsigned signal,		
`input` is a signal,		
`ram` is a `RAMw` signal.		
`ROM(content, address)`	ROM tabulating `content`	`Vector(1.5, 18.2)(address)`
`RAMw(input, address)`	Write port of a RAM	`val r = RAM(in, address1)`
`RAMr(ram, address)`	Read port of a RAM	`r(address2)`
Multiplexer signal		
`content` is an indexed sequence of signals of the same type,		
`address` is an unsigned signal.		
`Mux(content, address)`	Multiplexer	`Vector(rhs, lhs)(address)`
Bus manipulation signals		
`lhs` and `rhs` are signals of the same type,		
`range` is a range of integers.		
`Cons(lhs, rhs)`	Binary concatenation	`lhs ++ rhs`
`Tap(lhs, range)`	Extraction of a selection of bits	`lhs(range)`
Synchronization signals (provided by streaming blocks)		
`size` is an integer.		
`Timer(size)`	Number of cycles since the last dataset entered	`Timer(8)`
`Counter(size)`	Number of datasets that have been processed	`Counter(4)`

Table 9: Example of nodes (signals) used in the streaming-RTL DSL, and corresponding syntax.

one (flip/flop registers used for pipelining), or more parent signals (see Table 9). In the case of streaming reuse, this graph may contain loops.

The graph is constructed and represented using a *Streaming-RTL* DSL. Hardware datatypes, pipelining decisions and synchronization issues are mostly abstracted from this language. As an example, the implementation of the streaming block for the twiddles T_j can be written within a few lines, and works for every folding scenario and hardware datatype:

```
case class Twiddles(n: Int, k: Int, j: Sig[Int])(implicit dt: HW[Complex[Double]])
  extends SB[Complex[Double]](1 << n, 1 << k){
  override def implement(inputs: Vector[Sig[Complex[Double]]]) = {
    // We first declare a timer that ticks for the duration of a dataset
    val timer = Timer(1 << t)

    // we define a (non-staged) Vector containing all 2^n th roots of unity
    val rootsOfUnity = Vector.tabulate(1 << n){i =>
      val angle = -2 * Math.Pi * i / (1 << n)
      Complex(Math.cos(angle), Math.sin(angle))
    }

    // For each input signal,
    inputs.zipWithIndex.map{case (input, p) =>
      // we construct a signal corresponding to the index of a given element
      // (concatenation of the t bits of the timer, and the k bits of
      // the current port p),
      val i = timer ++ p(Unsigned(k))

      // we compute the corresponding twiddle factor,
      val address = (i & 1) * ((i >>> (j + 1)) << j)
      val twiddle = rootsOfUnity(address)

      // and we return the product of the input signal with this twiddle factor
      input * twiddle
} } }
```

As can be seen, only a few elements in the body of this function (`Timer`, `Unsigned`) may indicate that this code represents a low-level hardware architecture. This improves its readability and therefore its maintainability. However, all signals implicitly carry an underlying hardware type (including the corresponding size in bits), and timing information. All operations are bit- and cycle-accurate, and software and hardware type-safety is ensured. This DSL and its implementation are detailed in the next section.

Once constructed and optimized, the resulting graph is translated to a Verilog file.

5.2 A DSL FOR "STREAMING-RTL"

Our streaming-RTL DSL is used to construct from a streaming-block level representation of an algorithm a dependency graph that represents the final circuit. In this graph, the nodes (signals) represent hardware operators, and the edges the dependencies between these signals. The DSL offers the following features:

- The nodes (*signals*) of the graph are manipulated exactly as the values they would represent in a regular Scala program. Only their type changes.
- The language provides genericity over the actual hardware datatype and precision. However, the datatype can be made explicit, offering bit-accurate control.

- Pipelining and synchronization of data-independent control is performed implicitly, but timing information and manual pipelining remains available.

We discuss next the implementation of these abstractions, using the features offered by the Scala type system.

5.2.1 *Staging and LMS*

The implementation of our DSL uses the concept of *staging*, in particular as done in LMS [65], but using our own implementation. Staging allows to distinguish those parts of the computation to be evaluated at generation time and those that will be implemented in hardware via a type annotation. Specifically, staging is done by changing a type `T` to the type `Sig[T]`; the latter means that computations on this type will be delayed, and may become part of the hardware implementation.

For example, in the following code, the first line defines a function `f1` that yields the sum of its parameters (`x` and `y` of type `Double`) augmented by 18. The return type (`Double`) is inferred by the compiler. In the second line however, `f2` returns an expression tree representing the computation on symbolic inputs.

```
def f1(x: Double, y: Double) = x + y + 18
def f2(x: Sig[Double], y: Sig[Double]) = x + y + 18
```

This tree can then be translated (*unparsed*) to RTL-Verilog, yielding an implementation of two adders (adding two signals and an immediate).

This behavior is obtained through the class `Sig[T]`, which instances represent the nodes in an expression tree:

```
abstract class Sig[T:HW]{
  // timing information
  val delay: Option[Delay]

  // field containing the hardware representation
  val hw = implicitly[HW[T]]
}
```

This class takes as a *type parameter* the type `T` of the expression it represents. This type is expected to come along with a *hardware representation*, provided as a type class `HW` (see Section 5.2.2). Additionally, each node is expected to provide timing information through the field *delay* (see Section 5.2.3).

The different types of computation are represented by a class that inherits from `Sig`:

```
// Addition of two nodes
case class Plus[T](lhs: Sig[T], rhs: Sig[T]) extends Sig[T] {...}

// Node containing a constant
case class Const[T:HW](value: T) extends Sig[T] {
  override val delay = None
}

// Pipelining register
case class Register[T](input: Sig[T]) extends Sig[T]{
  override val delay = input.delay.map(_ + 1)
}

...
```

Each instance `Sig[T]` offers (*lifts*) the same operators as a regular instance of `T` would (see Section 5.2.4). These lifted operators return the corresponding node in the form of another instance of `Sig`.

5.2.2 *Abstraction over hardware datatypes*

Type classes [83, 53] are a form of static ad hoc polymorphism, that, contrary to inheritance, allows to retroactively add functionality to existing data types. For instance, the following function `f3` is generic in the type `T` of its parameters, but imposes that this type is numeric:

```
def f3[T:Numeric](x: T, y: T) = x * y
```

In Scala, type classes are implemented using regular classes: `f3` expects a third implicit argument of type `Numeric[T]` containing, among other, the definition of the operator `*` for two `T`s.

Following the concept of *abstraction over data representation* from [54], instances of `Sig[T]` (*signals* of `T`) carry their underlying hardware representation in the form of a type class `HW[T]`:

```
abstract class HW[T](val size: Int){
  // bit representation of a value
  def getBits(value: T): BigInt

  // creates a constant with this hardware representation
  def apply(value: T) = Const(value)(this)
}
```

Not only does this type class provide an additional method `getBits` that returns the bit representation of a given `T`, but it carries as meta-information the size in bits of the representation. Concrete hardware representations are instances of classes derived from `HW[T]`:

```
// Signed integer
case class Signed(_size: Int) extends HW[Int](_size){...}

// Unsigned integer
case class Unsigned(_size: Int) extends HW[Int](_size){...}

// Fixed point number
case class FxP(integral: Int, fractional: Int)
  extends HW[Double](integral + fractional){...}

// IEE754 floating point
case class IEEE(wE: Int, wF: Int) extends HW[Double](wE + wF + 1){...}

// FloPoCo floating point
case class FloPoCo(wE: Int, wF: Int) extends HW[Double](wF + wE + 3){...}
...
```

A Scala `Int` could therefore be represented as a signed or unsigned integer of a given size, and a Scala `Double` can be represented using a fixed-point representation, a FloPoCo number[2] or an IEEE 754 floating-point representation. This information is passed to children nodes, and is used for the implementation of the lifted operators and for the representation of constants in the generated code. As an example, depending on the underlying hardware type of its parameters, the previous example `f3` would seamlessly

- use fixed-point adders and represent 18 as a fixed-point immediate, or

2 The FloPoCo generator is called upon instantiation of the corresponding datatype class to generate the different arithmetic operators. The result of this generation is then parsed to extract the latency of these operators.

- use FloPoCo generated floating-point adders and represent 18 with the corresponding FloPoCo binary representation, or
- implement a conversion from an IEEE floating-point signal to a FloPoCo representation, implement the FloPoCo adder, and implement the conversion back to an IEEE representation.

5.2.3 *Synchronization*

Each signal has a *delay* field that represents the time needed for this signal to output a valid value. It is used to check if two operands are synchronized, and, if it is not the case, to suitably delay one of them using registers.

A delay consists of an integer representing a number of cycles, and a *timeline* indicating to which "reference frame" this delay belongs. This timeline can be

- the *primary* timeline, referring to the number of cycles elapsed since the inputs arrived in the module,
- a *loop* timeline, referring to the number of cycles elapsed since a dataset entered within a loop, or
- a *floating* timeline, used by data-independent signals awaiting to be "synchronized" with another timeline.

As an example, a `Register` would have the same delay as its input with a cycle number incremented by one, while an input signal would have a delay of `0` on the primary timeline.

Loop timelines. In the case of iterative reuse, the *streaming product* (the streaming block that creates the loop and the multiplexer in Figs. 2b, 2d and 26b) creates a new loop timeline, and implements its inner expression using this timeline. The corresponding latency is then measured using the maximal delay of the signals that are returned. This information is then used during a second unparsing of the inner expression, where a possible lack of latency is compensated by a FIFO, or an increase of latency of a potential inner temporal permutation. The streaming product then presents its outputs using the same timeline as its inputs, delayed accordingly.

Floating timelines. In our generator, all data-independent control signals rely on counters (that count the number of datasets that have passed) and on timers (that count the number of cycles elapsed since the beginning of the current dataset). To ensure that such control signals become available at the correct instant, each time a new counter or timer is declared, a corresponding floating timeline is created. All data-independent operations performed are then pipelined using this timeline. However, when a signal with a floating timeline and a signal with an external timeline need to be synchronized, a new *floating delay* node is inserted with the expected delay.

As an example, we consider the following function `f4`:

```
def f4(x: Sig[Int]) = {
  val t = Timer(8) + 3
  x ^ t
}
```

This function creates a 3-bit timer, and adds the constant 3 to it. This operation implicitly adds a pipelining register, yielding a signal `t` with a delay of 1 on the floating timeline associated with the timer. The input signal `x` is then xored with `t`. As these

two signals are associated with different timelines, a floating delay signal depending on `t` is created with the same `delay` member as `x`, and `f4` finally returns a signal representing a XOR of `x` and the floating delay signal.

After the graph construction, the floating timeline is synchronized with the other timeline such that all floating delays can be implemented using the minimal number of registers. In particular, this ensures that data-dependent signals never have to be uselessly delayed. In our example, the floating timeline is synchronized such that the floating delay is implemented with a direct assignment. Thus, a delay of one cycle on the floating timeline corresponds to the delay of `x`.

To prevent nodes of a floating timeline from being synchronized with different incompatible timelines, and to avoid circular dependencies between floating timelines, the first time a node of a floating timeline is synchronized with a node from another timeline, the floating timeline is marked as "being in translation" with this other timeline, and an error is thrown if a node is later synchronized with a third timeline. With this relation, when the graph is built, timelines form a set of trees, rooted by the primary and loop timelines. Floating timelines are then synchronized starting from the roots.

Synchronization tokens. When the graph is unparsed, token synchronization signals are generated to trigger the different counters and timers. Tokens for loop timelines are generated by "ORing" tokens of the primary timeline. As the maximal throughput of the design is known at this time, tokens of the primary timeline can be generated using consecutive resettable timers instead of a resettable shift-register.

In our previous example, the timer declared within `f4` receives its token one cycle before `x` becomes available, ensuring that `t` is computed at the right time.

5.2.4 *Smart constructors*

Lifted operators are provided using *implicit classes*, which make it possible to add a posteriori methods and operators to existing objects.

For instance, the following class provides a + operator to any `Sig[T]`, when `T` is a numeric type:

```
implicit class NumericSig[T:Numeric](lhs: Sig[T]){
  // the default hardware representation when creating Const is the one of
  // the left-hand side
  implicit val hw = lhs.hw

  // lhs + rhs in the case where lhs is a Sig[T] and rhs a T
  def +(rhs: T): Sig[T] = lhs + Const(rhs)

  // lhs + rhs in the case where both lhs and rhs are Sig[T]
  def +(rhs: Sig[T]): Sig[T] = {
    // check if lhs and rhs have the same representation
    ensure(rhs.hw == hw)
    (lhs, rhs).synch match {
      // both lhs and rhs are Const
      case (Const(x), Const(y)) => Const(x + y)

      // one of the operands is null
      case (Zero(), _) => rhs
      case (_, Zero()) => lhs

      // otherwise, we create a new node specialized for
      // the hardware representation
      case (lhs, rhs) => hw match {
        case _: FloPoCo => PlusFPC(lhs, rhs).register
        case _: IEEE => (lhs.toFPC + rhs.toFPC).toIEEE
        case _: Signed | _: Unsigned | _: FxP => Plus(lhs, rhs).register
        case _ => throw new Exception("No hardware implementation for +")
} } } }
```

Here, the operator first checks that the two operands have the same hardware type (ensuring type-safety). It then synchronizes them, and handles particular cases (if the two operands are constants, or if one of them is the constant zero). Finally, it creates a new Plus signal, according to the hardware datatype, and adds pipelining registers (in the case of a FloPoCo operator, the signal `PlusFPC` already takes into account the latency of the operator. A final register is added. For signed, unsigned integers and fixed-point representations, the pipeline has always a depth of one cycle. It would however be possible to adapt it to the size of the operands, the target architecture or the target frequency. In case of an IEEE representation, the pipelining is handled by the underlying call to the FloPoCo operator.).

These *smart constructors* are responsible for major optimizations. As an example, the constructor of ROM signals (implemented by adding a new `apply` method on indexed sequences of `T`) checks every bit of the control signal, and returns a smaller ROM in the case where one of them is constant. Particularly, it would return a constant if the control signal is constant, thus guaranteeing an efficient implementation of the twiddle stage T_j, even in non-streaming or non-iterative cases.

5.3 STREAMING-BLOCK DSL

The streaming-block DSL is an intermediate language between SPL and the Streaming-RTL DSL (See Fig. 23). It supports optimizations at the streaming level, i.e. optimizations that take place once an algorithm has been "folded" according to a given streaming width.

5.3.1 *Streaming blocks*

Each streaming-block represents a hardware module that has the same number (`K`) of inputs and outputs of the same hardware type (`HW[T]`), and that performs an operation on a dataset of size `N`. Streaming-blocks are comparable to SPL elements with a streaming width information, and are instances of classes derived from `SB`:

```
abstract class SB[T:HW] (N: Int, K: Int){
  // Size must be a multiple of the streaming width
  assert(N % K == 0)

  def implement(inputs: Vector[Sig[T]]): Vector[Sig[T]]
  // Composition of blocks
  def *(rhs: SB[T]) = Product(this, rhs)
}
```

Streaming blocks are expected to override a virtual method `implement` that constructs the final circuit using the streaming-RTL DSL[3]. An operator $*$ allows to compose blocks, as explained in Section 5.3.2 below.

As an example, an array of butterflies (corresponding to the SPL expression $\bigoplus \mathrm{DFT}_2$) would be implemented as follows:

```
case class ButterflyArray[T:HW:Numeric] (_N: Int, _K: Int) extends SB[T](_N, _K){
  // The streaming width must be even
  assert(sw % 2 == 0)

  override def implement(inputs: Vector[Sig[T]]) = {
    assert(inputs.size == K)
    // Compute the sum and difference of each pair of inputs
    inputs.grouped(2).toVector.flatMap{case Vector(a, b) => Vector(a + b, a - b)}
} }
```

5.3.2 *Higher-order blocks*

Composition of blocks is achieved through the block `Product`.

```
case class Product[T:HW] private (factors: Vector[SB[T]]) extends
  SB[T](factors.head.N, factors.head.K){
  // Size and streaming width of all factors must be the same
  assert(factors.forall(f => f.N == N && f.K == K))

  override def implement(inputs: Vector[Sig[T]]) = {
    assert(inputs.size == K)
    // Implement all factors by connecting the outputs of one to the inputs of
    // the next
    factors.foldRight(inputs)((sb, cur) => sb.implement(cur))
} }
```

A companion object of `Product` contains a smart constructor (this is the one called by the operator $*$ in `SB`), along with a higher-order function that implements the block corresponding to the SPL expression $\prod_{j=0}^{\text{limit}} f(j)$.

```
object Product{
  def apply[T:HW](lhs: SB[T], rhs: SB[T]): Product[T] = (lhs, rhs) match {
      case (Product(f1), Product(f2)) => new Product(f1 ++ f2)
      case (Product(f1), _) => new Product(f1 :+ rhs)
      case (_, Product(f2)) => new Product(lhs +: f2)
      case _ => new Product(Vector(lhs, rhs))
```

3 The real implementation adds an option to optionally add latency, and returns the minimal number of cycles that the circuit can handle between datasets (gap).

```
    }

  def apply[T:HW](limit: Int)(f: Sig[Int] => SB[T]): SB[T] = {
    assert(limit > 0)
    // Unsigned with the minimal size that can contain all j
    val idxType = Unsigned(BigInt(limit - 1).bitSize)
    (0 until limit).map(i=>Const(i)(idxType)).map(f).reduce(_*_)
} }
```

Note that the higher-order function expects a function that returns a block, and that takes an integer signal as a parameter (and not directly an integer). This allows to have another block, `ItProduct` with the same interface, but that produces a loop for iterative reuse (by implementing a multiplexer, creating a new loop timeline, and calling `f` with a `Counter` as a parameter).

The streaming-block DSL does not directly have any operator corresponding to the direct sum of SPL: $\bigoplus_j f(j)$. Before folding, the SPL expression must therefore have all the direct sums fully distributed. Then, the remaining direct sums must be handled within the streaming blocks themselves, as it is the case with `ButterflyArray`.

5.3.3 *Permutation blocks*

Apart from the direct sum operator, permutations are the only SPL operators that do not have a direct equivalent in the streaming-block DSL. Only two types of permutations are directly implementable as streaming-blocks:

- *Spatial* permutations: these are permutations that permute elements only within the same cycle. They can be implemented using switches (Fig. 11c), or multiplexers (Fig. 11b).
- *Temporal* permutations: these permutations permute elements only between cycles, but stay on the same port number. They can be implemented using an array of memory banks as in Fig. 11a.

During the folding operation, general permutations are decomposed into these using the algorithms described in Chapter 2.

5.3.4 *Optimizations*

The optimizations taking place at this step mainly concern streaming permutations and iterative reuse loops that have an early termination. They are described with more details in Chapter 3. For instance, a streaming permutation block within an iterative loop may be unrolled under certain conditions, thus reducing the global number of multiplexers used within the whole design, or increasing the throughput of the design. Another optimization consists in fusing two consecutive arrays of 2-input switches into an array of 4-input switches, which can improve the resource used on some FPGA architectures. Fig. 27 shows a case where these two optimizations were performed.

5.4 RESULTS

To validate the designs produced by our generator, we benchmarked them against the equivalent circuits generated with [47]. All designs were synthesized using Vivado

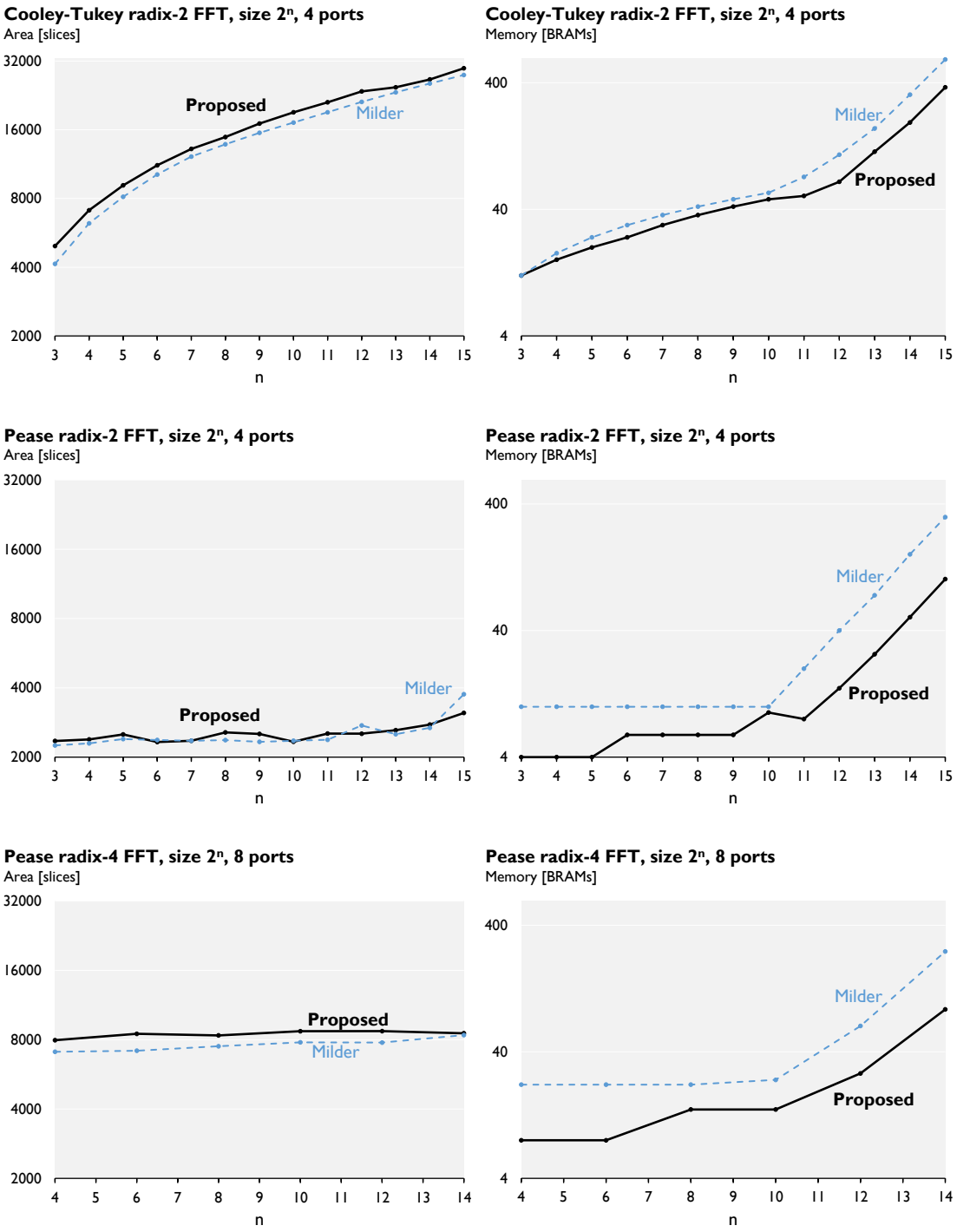


Figure 28: Resources used by different FFTs (40) in different configurations, on complex data using 2×32bits IEEE754 floating-point.

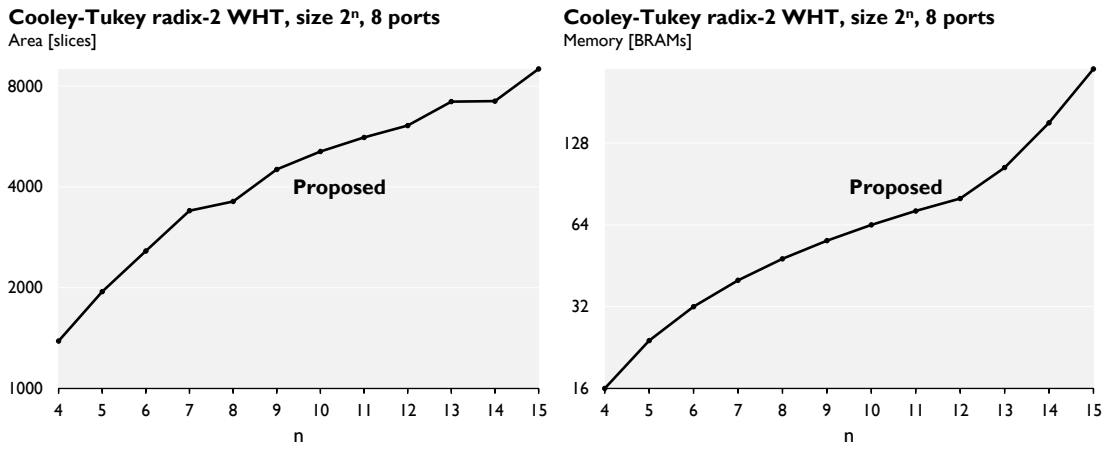


Figure 29: Resources used by a radix-2 Cooley-Tukey WHTs (41) with a streaming width of 8, on complex data using 2×32bits fixed-point.

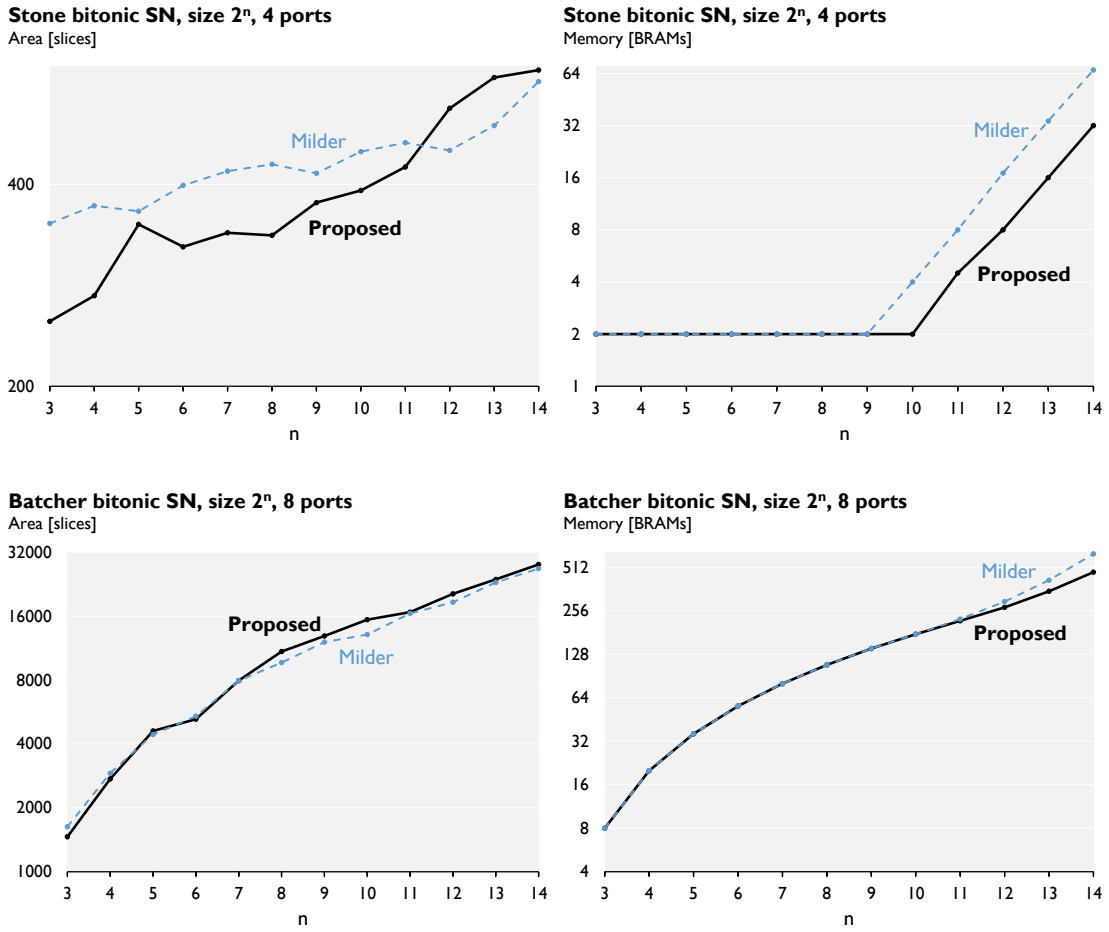


Figure 30: Resources used by different bitonic sorting networks (42), with different streaming configuration on complex data using 2×32bits fixed-point.

2018.1, targeting a Virtex7 xc7vx1140 FPGA. The floating-point operators used in our designs were generated using FloPoCo 4.1.2, targeting a 700MHz Virtex6 platform.

Figures 28, 29 and 30 show results after place-and-route for a variety of transforms, algorithms, hardware datatypes and foldings. Each of these presents, for a given transform size, the resources used in terms of logic slices and memory obtained for our design and the corresponding design from [47] or [89]. Cooley-Tukey FFTs and WHTs (Figs. 28a, 28b and 29) and Stone SNs (Figs. 30a and 30b) are implemented using only streaming reuse. Batcher SNs (Figs. 30c and 30d) use both streaming and iterative reuse. Pease FFTs (Figs. 28c, 28e, 28d and 28f) use streaming and iterative reuse with fused permutations, as described in Chapter 3.

Figs. 28d and 28f also show a comparison with the designs obtained using Xilinx IP generator Fast Fourier Transform 9.1. The parameters were set to resemble as much as possible our architecture. A radix-2 Burst IO Lite architecture is used with 4 channels (in Fig. 28d), and a radix-4 Burst IO is used with 8 channels (in Fig. 28f), in each case with a natural output ordering, and using 32 bit fixed point representation for both inputs and phase factors. However, as neither floating point computation nor the streaming IO architecture are available when using multiple channels, no comparison can be made for logic consumption, nor for the Cooley-Tukey architecture.

All our designs were generated with sufficient pipelining to reach the same frequencies as [47] or [89] (in the order of 400MHz). The throughput of these designs are therefore the same. Additionally, designs requiring complex multiplications (Fig. 28) use an algorithm that yield the same number of DSP slices as [47].

We observe that the logic area consumption slightly increases. The gain obtained using FloPoCo for the arithmetic part and the use of 4-input multiplexers is counterbalanced by the additional logic needed to implement memory conflict avoidance as described in [68].

On the other side, the number of BRAM tiles required is lower in average with our generator. This is a direct consequence of the streaming permutations being implemented with [68]. As it does not use double buffering, the capacity required to implement streaming permutations is halved. However, this reflects on the number of BRAMs only when the double buffer does not fit into a single BRAM, that is, for large sizes of n ($n > 10$ in Fig. 28 and $n > 9$ in Fig. 30b). In addition, in the case of FFTs with iterative reuse (Figs. 28d and 28f), the stride permutation and the bit-reversal are fused, allowing to halve the number of BRAMs used for streaming permutations. However, these techniques do not affect the number of RAM slices used as ROMs to store the twiddle factors. Finally, both the designs we propose and those generated using [47] require significantly fewer RAMs than Xilinx Fast Fourier transform IP cores.

In summary, our generator produces designs that use an equal amount or less memory than [47] or [89], particularly for large sizes, for the same number of DSP blocks, and for a comparable area consumption. Our generator is thus able to improve the state of the art for important parts of the design space of the considered transforms, and yields new Pareto optima.

5.5 LIMITATIONS AND RELATED WORK

We compare to related work and discuss limitations.

5.5.1 *Hardware DSLs implemented in Scala*

The DSL we propose is specifically crafted for the generation of streaming Fourier transforms and sorting networks on FPGAs, and provides only the primitives and the amount of abstraction needed for this purpose. This differentiates it from lower level hardware description languages written in Scala. For instance, Chisel [3] can represent a much wider variety of hardware designs, but requires the pipelining registers to be manually added. Targeting dataflow hardware, DFiant [62] proposes a dependency-driven automatic pipelining similar to ours, but does not seem to support automatic synchronization of data-independent controls. It uses literal types to expose the hardware datatype and precision to the user, thus enforcing type safety at compile-time. In our case, the hardware datatype is abstracted (provided via a type class), and hardware type safety is only ensured at generation time. On the other hand, high-level synthesis tools [80, 28] would offer even higher abstractions, up to the dataset level, but would not allow the user to program at the port-level, thus making the implementation of our permutation streaming blocks difficult.

LMS [65] itself not only provides staging, but offers a tool chain to implement and compile DSLs. Particularly, it grants automatic common subexpression elimination during the construction of the dependency graph. However, in our case, floating timelines reduce the efficiency of such an optimization during the graph construction, and our tests have shown that synthesis software such as Vivado already provide it, thus limiting the use of implementing it. LMS provides as well a facility to manipulate the generated graph, but as ours already includes timing information, these manipulations are limited to timing invariant ones (fusing ROMs that contain identical values for instance), for which a direct implementation is possible. The main optimizations in our graph are made during its generation, using smart constructors.

The pipeline proposed in [27] to generate matrix operations illustrates the capability of LMS to target hardware. It shares many similarities with ours, particularly its use of LMS and FloPoCo. However, a significant part of the final RTL design is outsourced to the external back-end LegUp [8].

5.5.2 *Hardware generator for FFTs*

Our generator only handles the generation of power-of-two-sized FFTs, whereas [47] covers a larger set of sizes that can be factored into small primes and additional transforms closely related to FFTs. An according extension of our generator should be relatively straightforward. Note that [47] works as a back-end of Spiral [64], a generator written on a modified version of the GAP computer algebra system, thus requiring high skills for its development.

SPL and Spiral have as well been implemented and enhanced in Haskell [42] and in Scala [52] to produce efficient FFT implementations in C. A VHDL back-end for this compiler is being developed [42].

5.5.3 *Hardware generator for sorting networks*

The work in [88, 89] presents a generator for streaming sorting networks, and we have shown how our generator outperform its RAM consumption for the algorithms we support. However, [88, 89] covers a larger space of algorithms (called SN2–4), and was

originally targeting (and thus optimized for) a platform (Xilinx Virtex 5) older than the one we used for our benchmarks (Xilinx Virtex 7).

Sorting is a classic topic in computer science [4, 37], and many high-performance sorting networks have been manually implemented on FPGAs, using 2-inputs sorters [48, 12], or using other basis elements like linear sorters [56].

5.6 CONCLUSION

In this chapter, we have designed and implemented a generator for streaming FFTs and sorting networks inside Scala, using embedded DSLs and the concept of staging. It followed a principled design of domain-specific hardware generators using state-of-the-art languages and language features.

Specifically, our generator employed a pipeline of three abstraction levels, corresponding to three levels of DSLs. Two of them, the streaming-block DSL and the streaming-RTL DSL are novel and were specifically designed to include state-of-the-art components and enable the transformations and optimizations needed in FFTs. The generator should be easily extendable to other DSP components related to FFTs. A web version of our generator is available at [67], and its source code at [66].

Part III

LINEAR ALGEBRA THEOREMS

6

A LOWER-UPPER-LOWER BLOCK TRIANGULAR DECOMPOSITION WITH MINIMAL OFF-DIAGONAL RANKS

In Chapter 2, we showed that the optimal implementation of an SLP $(\pi(P), 2^k)$ reduces to a matrix decomposition problem: factor the invertible bit matrix

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} = \begin{pmatrix} I_t & \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & C_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R_2 & R_1 \end{pmatrix},$$

such that $\operatorname{rank} L_2 + \operatorname{rank} R_2$ is minimal. The existence of this decomposition and the minimum was stated in Theorem 2.

In this chapter, published in [69], we provide a proof of this theorem for an invertible matrix P over any field. The factorization decomposes P into a product of three matrices that are lower block-unitriangular, upper block-triangular, and lower block-unitriangular, respectively. In words, our goal is to make this factorization "as block-diagonal as possible" by minimizing the ranks of the off-diagonal blocks. We give lower bounds on these ranks and show that they are sharp by providing an algorithm that computes an optimal solution. The proposed decomposition can be viewed as a generalization of the well-known Block LU factorization using the Schur complement.

Theorem 2 uses this factorization for bit matrices, i.e., matrices over the Galois field $\mathbb{F}_2$. However, we believe that because of its simple and natural structure, the matrix decomposition is also of pure mathematical interest, and our proof only assumes a field $\mathbb{K}$. Our algorithm computes an optimal solution with an asympotic number of operations cubic in the matrix size. We implemented the algorithm for finite fields, for rational numbers, for Gaussian rational numbers and for exact real arithmetic for validation. For a floating point implementation, numerical issues may arise.

6.1 PROBLEM STATEMENT

Given is a non-singular (invertible) matrix $P \in \mathrm{GL}_{t+k}(\mathbb{K})$ over a field $\mathbb{K}$. We partition P as

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix}, \quad \text{such that } P_4 \text{ is } t \times t.$$

We denote the ranks of the submatrices with $p_i = \operatorname{rank} P_i$, $i = 1, 2, 3, 4$. Matrices are denoted with capital letters and vector spaces with calligraphic letters.

If P_4 is non-singular, then a block Gaussian elimination uniquely decomposes P into the form:

$$P = \begin{pmatrix} I_t & \\ L & I_k \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & C_1 \end{pmatrix}, \tag{46}$$

where I_t denotes the $t \times t$ identity matrix. The rank of $L = P_2 P_4^{-1}$ is equal to p_2, and C_1 is the Schur complement of P_4. Conversely, if such a decomposition exists for P, then P_4 is non-singular. This block LU decomposition has several applications including

computing the inverse of P [13], solving linear systems [18]. The Schur complement is also used in statistics, probability and numerical analysis [87, 17].

Analogously, the following decomposition exists if and only if P_1 is non-singular:

$$P = \begin{pmatrix} C_4 & C_3 \\ & C_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R & I_k \end{pmatrix}. \tag{47}$$

This decomposition is again unique, and the rank of R is p_2.

In this article, we release the restrictions on P_1 and P_4 and propose the following decomposition for a general $P \in \mathrm{GL}_{t+k}(\mathbb{K})$:

$$P = \begin{pmatrix} I_t & \\ L & I_k \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & C_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R & I_k \end{pmatrix}, \tag{48}$$

where in addition we want the three factors to be "as block-diagonal as possible," i.e., that $\operatorname{rank} L + \operatorname{rank} C_3 + \operatorname{rank} R$ is minimal.

6.1.1 *Lower bounds*

The following theorem provides bounds on the ranks of such a decomposition:

Theorem 4. *If a decomposition* (48) *exists for* $P \in \mathrm{GL}_{t+k}(\mathbb{K})$*, it satisfies*

$$\operatorname{rank} C_3 = p_3, \tag{49}$$
$$\operatorname{rank} L \geqslant k - p_1, \tag{50}$$
$$\operatorname{rank} R \geqslant t - p_4, \tag{51}$$
$$\operatorname{rank} R + \operatorname{rank} L \geqslant p_2. \tag{52}$$

In particular, the rank of C_3 *is fixed and we have:*

$$\operatorname{rank} R + \operatorname{rank} L \geqslant \max(p_2, k + t - p_1 - p_4). \tag{53}$$

We will prove this theorem in Section 6.3. Next, we assert that these bounds are sharp.

6.1.2 *Optimal solution*

The following theorem shows that the inequality (53) is sharp:

Theorem 5. *If* $P \in \mathrm{GL}_{t+k}(\mathbb{K})$*, then there exists a decomposition* (48) *that satisfies*

$$\operatorname{rank} R + \operatorname{rank} L = \max(p_2, k + t - p_1 - p_4) \text{ and } \operatorname{rank} L = k - p_1.$$

Additionally, such a decomposition can be computed with $O((t+k)^3)$ *arithmetic operations.*

We prove this theorem in Section 6.4 when $p_2 \leqslant t + k - p_1 - p_4$, and in Section 6.5 for the case $p_2 > t + k - p_1 - p_4$. In both cases, the proof is constructive and we provide a corresponding algorithm (Algorithms 3 and 4). Theorem 5 and the corresponding algorithms are the main contributions of this article.

Two cases. As illustrated in Figure 31, two different cases appear from inequality (53). If $p_2 \leqslant t + k - p_1 - p_4$, bound (52) is not restrictive, and the optimal pair

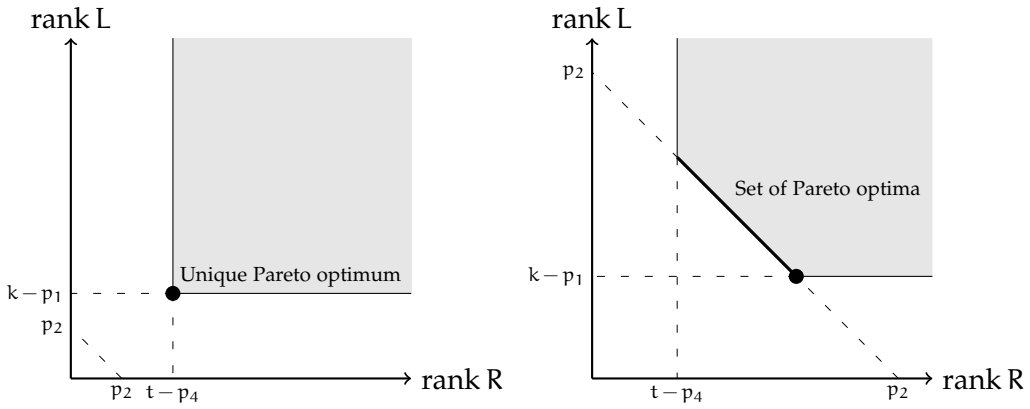


Figure 31: Possible ranks for L and R. On the left graph, $p_2 < t+k-p_1-p_4$. On the right graph, $p_2 > t+k-p_1-p_4$. The dot shows the decomposition provided by Theorem 5.

$(\operatorname{rank} L, \operatorname{rank} R)$ is unique, and equals $(k-p_1, t-p_4)$. In the other case, where $p_2 > t+k-p_1-p_4$, bound (52) becomes restrictive, and several optimal pairs $(\operatorname{rank} L, \operatorname{rank} R)$ exist.

Example. As a simple example we consider the special case

$$P = \begin{pmatrix} & P_3 \\ P_2 & \end{pmatrix}, \quad \text{with } k = t.$$

In this case P_2, P_3 are non-singular and neither (46) nor (47) exists. Theorem 4 gives a lower bound of $\operatorname{rank} R + \operatorname{rank} L \geqslant 2k$, which implies that both R and L have full rank. Straightforward computation shows that for any non-singular L,

$$P = \begin{pmatrix} I_k & \\ L & I_k \end{pmatrix} \begin{pmatrix} L^{-1}P_2 & P_3 \\ & -LP_3 \end{pmatrix} \begin{pmatrix} I_k & \\ -(LP_3)^{-1}P_2 & I_k \end{pmatrix}$$

is an optimal solution. This also shows that the optimal decomposition (48) is in general not unique.

6.1.3 *Flexibility*

The following theorem adds flexibility to Theorem 5 and shows that a decomposition exists for any Pareto-optimal pair of non-diagonal ranks that satisfies the bounds of Theorem 4:

Theorem 6. *If* $P \in \mathrm{GL}_{t+k}(\mathbb{K})$ *and* $(l, r) \in \mathbb{N}^2$ *satisfies* $l \geqslant k-p_1$, $r \geqslant t-p_4$, *and* $r+l = \max(p_2, k+t-p_1-p_4)$, *then* P *has a decomposition* (48) *with* $\operatorname{rank} L = l$ *and* $\operatorname{rank} R = r$.

In the case where $p_2 \leqslant t+k-p_1-p_4$, the decomposition produced by Theorem 5 has already the unique optimal pair $(\operatorname{rank} L, \operatorname{rank} R) = (k-p_1, t-p_4)$. In the other case, we will provide a method in Section 6.6 to trade between the rank of R and the rank of L, until bound (51) is reached. By iterating this method over the decomposition obtained in Theorem 5, decompositions with various rank tradeoffs can be built.

Therefore, it is possible to build decomposition (48) for any pair $(\operatorname{rank} L, \operatorname{rank} R)$ that is a Pareto optimum of the given set of bounds. As a consequence, if $f : \mathbb{N}^2 \to \mathbb{R}$

is weakly increasing in both of its arguments, it is possible to find a decomposition that minimizes $f(\operatorname{rank} L, \operatorname{rank} R)$. Examples include $\min(\operatorname{rank} L, \operatorname{rank} R)$, $\max(\operatorname{rank} L, \operatorname{rank} R)$, $\operatorname{rank} L + \operatorname{rank} R$, $\operatorname{rank} L \cdot \operatorname{rank} R$ or $\sqrt{\operatorname{rank}^2 L + \operatorname{rank}^2 R}$.

Generalization of block LU factorization. In the case where P_1 is non-singular, Theorem 5 provides a decomposition that satisfies $\operatorname{rank} L = 0$. In other words, it reduces to the decomposition (47). Using Theorem 6, we can obtain a similar result in the case where P_4 is non-singular. Since in this case $t - p_4 = 0$, it is possible to choose $r = 0$, and thus obtain the decomposition (46).

6.1.4 *Equivalent formulations*

Lemma 1. *The following decomposition is equivalent to decomposition (48), with analogous constraints for the non-diagonal ranks:*

$$P = \begin{pmatrix} I_t & \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ R_2 & R_1 \end{pmatrix} \tag{54}$$

In this case, the minimization of the non-diagonal ranks is exactly the same problem as in (48). However, an additional degree of freedom appears: any non-singular $k \times k$ matrix can be chosen for either L_1 or R_1.

It is also possible to decompose P into two matrices, one with a non-singular leading principal submatrix L_4 and the other one with a non-singular lower principal submatrix R_1:

$$P = \begin{pmatrix} L_4 & L_3 \\ L_2 & L_1 \end{pmatrix} \begin{pmatrix} R_4 & R_3 \\ R_2 & R_1 \end{pmatrix} \tag{55}$$

Once again, the minimization of $\operatorname{rank} L_2 + \operatorname{rank} R_2$ is the same problem as in (48). The two other non-diagonal blocks satisfy $\operatorname{rank} L_3 + \operatorname{rank} R_3 \geqslant p_3$.

Proof. The lower non-diagonal ranks are invariant through the following steps:

(48) ⇒ (54)**.** If P has a decomposition (48), a straightforward computation shows that:

$$P = \begin{pmatrix} I_t & \\ L & C_1 \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ R & I_k \end{pmatrix},$$

which has the form of decomposition (54).

(54) ⇒ (55)**.** If P has a decomposition (54), the multiplication of the two left factors leads to formulation (55). In fact, $L_4 = C_4$ and R_1 are both non-singular.

(55) ⇒ (48)**.** If P has a decomposition (55), then using (46) on the left factor, and (47) on the right factor, and multiplying the two central matrices leads to formulation (48). □

6.1.5 *Related work*

Schur complement. Several efforts have been made to adapt the definition of Schur complement in the case of general P_4 and P_1. For instance, it is possible to define an indexed Schur complement of another non-singular principal submatrix [87], or use pseudo-inverses [9] for matrix inversion algorithms.

Alternative block decompositions. A common way to handle the case where P_4 is singular is to use a permutation matrix B that reorders the columns of P such that the new principal upper submatrix is non-singular [87]. Decomposition (46) then becomes:

$$P = PBB^T = \begin{pmatrix} I_t & \\ L & I_k \end{pmatrix} \begin{pmatrix} C_4 & C_3 \\ & C_1 \end{pmatrix} B^T$$

However, B needs to swap columns with index $\geqslant t$; thus B^T does not have the required form considered in our work.

One can modify the above idea to choose B such that B^{-1} has the shape required by decomposition (48):

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} \begin{pmatrix} I_t & \\ -R & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ R & I_k \end{pmatrix} = \begin{pmatrix} P_4 - RP_3 & P_3 \\ P_2 - RP_1 & P_1 \end{pmatrix} \begin{pmatrix} I_t & \\ R & I_k \end{pmatrix}$$

Then the problem is to design R such that $P_4 - RP_3$ is non-singular and $\operatorname{rank}(P_2 - RP_1) + \operatorname{rank} R$ is minimal. This basic idea is used in [63], where, however, only $\operatorname{rank} R$ is minimized, which, in general, does not produce optimal solutions for the problem considered here.

Finally, our decomposition also shares patterns with a block Cholesky decomposition, or the Block LDL decomposition, in the sense that they involve block unitriangular matrices. However, the requirements on P and the expectations on the decomposition are different.

6.2 PRELIMINARIES

In this section, we will prove some basic lemmas that we will use throughout this chapter.

6.2.1 *Properties of the blocks of an invertible matrix*

In this subsection, we derive some direct consequences of the invertibility of P on the range and the nullspace of its submatrices.

Lemma 2. *The following properties are immediate from the structure of* P*:*

$$\ker P_1 \cap \ker P_3 = \{0\} \tag{56}$$
$$\ker P_2 \cap \ker P_4 = \{0\} \tag{57}$$
$$\operatorname{im} P_1 + \operatorname{im} P_2 = \mathbb{K}^k \tag{58}$$
$$\operatorname{im} P_3 + \operatorname{im} P_4 = \mathbb{K}^t \tag{59}$$
$$P_2(\ker P_4) \cap P_1(\ker P_3) = \{0\} \tag{60}$$
$$P_4(\ker P_2) \cap P_3(\ker P_1) = \{0\} \tag{61}$$

Proof. We prove here equation (60). If $x \in \ker P_4$ and $y \in \ker P_3$ satisfy $P_2 x = P_1 y$, we have:

$$\begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} \cdot \begin{pmatrix} x \\ -y \end{pmatrix} = \begin{pmatrix} P_4 x - P_3 y \\ P_2 x - P_1 y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Since P is non-singular, $x = y = 0$, as desired. □

Operation related to subspaces	Matrix operation	Correspondence
Kernel of a matrix M	$\overline{\ker}\, M$	$\langle \overline{\ker}\, M \rangle = \ker M$
Direct sum of subspaces $\mathcal{A}, \mathcal{B}$	$\begin{pmatrix} A & B \end{pmatrix}$	$\begin{pmatrix} A & B \end{pmatrix} = \mathcal{A} \oplus \mathcal{B}$
Intersection of subspaces $\mathcal{A}, \mathcal{B}$	$A \,\overline{\cap}\, B$	$\langle A \,\overline{\cap}\, B \rangle = \mathcal{A} \cap \mathcal{B}$
Complement of a subspace $\mathcal{B}$ in $\mathcal{A}$	$A \,\overline{\ominus}\, B$	$\langle A \,\overline{\ominus}\, B \rangle \oplus \mathcal{B} = \mathcal{A}$

Table 10: We summarize matrix operators to perform basic operations on subspaces. M is a general matrix, while A and B are matrices with m rows that represent the subspaces $\mathcal{A}$ and $\mathcal{B}$; i.e. $\mathcal{A} = \langle A \rangle$ and $\mathcal{B} = \langle B \rangle$. Inspection of these routines shows that they all can be implemented with $O(m^3)$ runtime.

These equalities yield the dimensions of the following subspaces:

Corollary 5.

$$\dim P_3(\ker P_1) = \dim \ker P_1 = k - p_1 \tag{62}$$
$$\dim P_4(\ker P_2) = \dim \ker P_2 = t - p_2 \tag{63}$$
$$\dim P_1(\ker P_3) = \dim \ker P_3 = k - p_3 \tag{64}$$
$$\dim P_2(\ker P_4) = \dim \ker P_4 = t - p_4 \tag{65}$$
$$\dim \operatorname{im} P_1 \cap \operatorname{im} P_2 = p_1 + p_2 - k \tag{66}$$
$$\dim \operatorname{im} P_3 \cap \operatorname{im} P_4 = p_3 + p_4 - t \tag{67}$$

Proof. We prove equation (62). Since, by (56), $\ker P_1 \cap \ker P_3 = \{0\}$, the dimension of the image of $\ker P_1$ under P_3 has the same dimension as $\ker P_1$, which is $k - p_1$.

Equation (66) is a consequence of equation (58) and of the rank-nullity theorem. □

6.2.2 *Algorithms on linear subspaces in matrix form*

The algorithms we present in this chapter heavily rely on operations on subspaces of $\mathbb{K}^m$. To make the representation of these algorithms more practical for implementation, we introduce a matrix representation for subspaces and formulate the subspace operations needed on this representation.

We represent a linear subspace as an associated matrix[1] whose columns form a basis of this subspace. In other words, if $\mathcal{A}$ is a subspace of $\mathbb{K}^m$ of dimension n, then we represent it using a $m \times n$ matrix A such that $\operatorname{im} A = \mathcal{A}$. In this case, and only in this case, we will use the notation $\langle A \rangle = \operatorname{im} A$ to emphasize that the columns of A form a linear independent set.

With this notation we can formulate subspace computations as computations on their associated matrices as explained in the following. To formally emphasize this correspondence, these operations on matrices will carry the same symbol as the subspace computation (e.g., $\cap$) they represent augmented with an overline (e.g., $\overline{\cap}$). The operations are collected in Table 10. All algorithms in this chapter are written as sequences of these matrix operations. Because of this, we implemented the algorithms by first designing an object oriented infrastructure that provides these operations. Then we could directly map the algorithms, as they are formulated, to code.

Direct sum of two subspaces. If $\langle A \rangle, \langle B \rangle \leqslant \mathbb{K}^m$ are two subspaces, then the direct sum can be computed by concatenating the two matrices: $\langle A \rangle \oplus \langle B \rangle = \langle \begin{pmatrix} A & B \end{pmatrix} \rangle$.

1 We allow the existence of matrices with 0 column to represent the trivial subspace of $\mathbb{K}^m$.

Null space of a matrix. Gaussian elimination can be used to compute the null space of a given $m \times n$ matrix M. Indeed, if the reduced column echelon form of the matrix $\begin{pmatrix} M \\ I_n \end{pmatrix}$ is blocked into the form $\begin{pmatrix} M_4 & \\ M_2 & M_1 \end{pmatrix}$, where M_4 has m rows and no zero column, then $\ker M = \langle M_1 \rangle$. We denote this computation with $\overline{\ker} M = M_4$.

Intersection of two subspaces. For two subspaces $\operatorname{im} A, \operatorname{im} B \leqslant \mathbb{K}^m$, the intersection can be computed using the Zassenhaus algorithm. Namely, if the reduced column echelon form of $\begin{pmatrix} A & B \\ A & \end{pmatrix}$ is blocked into the form $\begin{pmatrix} C_4 & \\ C_2 & C_1 \end{pmatrix}$, where C_4 and C_1 have m rows and no zero column, then $\operatorname{im} A \cap \operatorname{im} B = \langle C_1 \rangle$. We denote this computation with $A \overline{\cap} B = C_4$.

Complement of a subspace within another one. Let $\operatorname{im} A \leqslant \operatorname{im} B \leqslant \mathbb{K}^m$. Then, a complement $\langle C \rangle$ of $\operatorname{im} A$ in $\operatorname{im} B$, i.e., a space that satisfies $\langle C \rangle \oplus \operatorname{im} A = \operatorname{im} B$, can be obtained as described in Algorithm 1[2]. We denote this operation with $C = B \overline{\ominus} A$.

Algorithm 1 Complement of a subspace within another

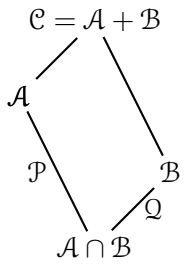
Require: Two matrices A and B with m rows
Ensure: A matrix $A \overline{\ominus} B$ such that $\langle A \overline{\ominus} B \rangle \oplus \operatorname{im} B = \operatorname{im} A$
  $C \leftarrow$ A $m \times 0$ matrix
  for each column vector b of B **do**
    if $\operatorname{rank} \begin{pmatrix} A & C & b \end{pmatrix} > \operatorname{rank} \begin{pmatrix} A & C \end{pmatrix}$ **then**
      $C \leftarrow \begin{pmatrix} C & b \end{pmatrix}$
    end if
  end for
  return C

6.2.3 *Double complement*

Lemma 3. *Let $\mathcal{C}$ be a finite-dimensional vector space and let $\mathcal{A}, \mathcal{B} \leqslant \mathcal{C}$ with $\dim \mathcal{A} \geqslant \dim \mathcal{B}$. Then, there exists a space $\mathcal{S} \leqslant \mathcal{C}$ such that:*

$$\begin{cases} \mathcal{S} \oplus \mathcal{A} = \mathcal{C} \\ \mathcal{S} \cap \mathcal{B} = \{0\} \end{cases}$$

Proof. We first consider the case where $\mathcal{C} = \mathcal{A} + \mathcal{B}$. We denote with $\mathcal{P}$ (resp. $\mathcal{Q}$) a complement of $\mathcal{A} \cap \mathcal{B}$ in $\mathcal{A}$ (resp. in $\mathcal{B}$).



2 This algorithm can be implemented to run in a cubic arithmetical time by keeping a reduced column echelon form of $\begin{pmatrix} A & C \end{pmatrix}$, which makes it possible to check the condition within the loop in a squared arithmetical time.

We show first that $\mathcal{P} \cap \mathcal{Q} = \{0\}$. Let $v \in \mathcal{P} \cap \mathcal{Q}$. As $\mathcal{P} \leqslant \mathcal{A}$ and $\mathcal{Q} \leqslant \mathcal{B}$, we have $v \in \mathcal{A} \cap \mathcal{B}$. Therefore, $v \in \mathcal{P} \cap \mathcal{A} \cap \mathcal{B} = \{0\}$, as desired.

We now denote with $b = \{b_1, \ldots, b_p\}$ (resp. $b' = \{b'_1, \ldots, b'_q\}$) a basis of $\mathcal{P}$ (resp. $\mathcal{Q}$), implying $q \leqslant p$. Considering $w = \{b_1 + b'_1, \ldots, b_q + b'_q\}$, the following holds:

i) w is linear independent: if $\{\alpha_1, \ldots, \alpha_q\} \in \mathbb{K}^q$ is such that $\sum \alpha_i w_i = 0$, then $\sum \alpha_i b_i = -\sum \alpha_i b'_i$. As $\sum \alpha_i b_i \in \mathcal{P}$ and $\sum \alpha_i b'_i \in \mathcal{Q}$, it comes that $\sum \alpha_i b'_i = \sum \alpha_i b_i = 0$. It follows that for all i, $\alpha_i = 0$, yielding the result.

ii) $\langle w \rangle \cap \mathcal{A} = \{0\}$: If $v \in \langle w \rangle \cap \mathcal{A}$, then there exists $\{\alpha_1, \ldots, \alpha_q\} \in \mathbb{K}^q$ such that $v = \sum \alpha_i (b_i + b'_i) \in \mathcal{A}$. It implies $\sum \alpha_i b'_i \in \mathcal{A}$. As the left hand side is in $\mathcal{Q}$, it comes that $\sum \alpha_i b'_i = 0$. It follows that for all i, $\alpha_i = 0$, yielding the result.

iii) $\langle w \rangle \cap \mathcal{B} = \{0\}$: Same proof as above.

Then, since $(\mathcal{A} \cap \mathcal{B}) \oplus \mathcal{Q} = \mathcal{B}$, we have $\dim \mathcal{C} = \dim A + \dim B - \dim(\mathcal{A} \cap \mathcal{B}) = \dim A + q$. Therefore, $\dim\langle w \rangle = q = \dim \mathcal{C} - \dim \mathcal{A}$, and $\mathcal{S} = \langle w \rangle$ satisfies the desired conditions.

In the general case, where $\mathcal{C} > \mathcal{A} + \mathcal{B}$, we use the same method, and simply add a complement $\mathcal{S}'$ of $\mathcal{A} + \mathcal{B}$ in $\mathcal{C}$ to the solution.

□

Algorithm 2 uses the method in this proof to compute a basis of $\mathcal{S}$, given $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. Note that if $\dim \mathcal{A} = \dim \mathcal{B}$, then $\mathcal{S}$ is a complement of both $\mathcal{A}$ and $\mathcal{B}$ in $\mathcal{C}$.

Algorithm 2 "Double complement" algorithm (Lemma 3)

Require: A, B and C, such that $\operatorname{im} A, \operatorname{im} B \leqslant \operatorname{im} C$ and $\operatorname{rank} A \geqslant \operatorname{rank} B$
Ensure: A matrix S such that $\langle S \rangle \oplus \operatorname{im} A = \operatorname{im} C$ and $\langle S \rangle \cap \operatorname{im} B = \{0\}$.
$P \leftarrow A \overline{\ominus} (A \overline{\cap} B)$
$Q \leftarrow B \overline{\ominus} (A \overline{\cap} B)$
$P' \leftarrow P$ truncated such that P' and Q have the same size
$S' \leftarrow C \overline{\ominus} \begin{pmatrix} A & B \end{pmatrix}$
return $\begin{pmatrix} S' & P' + Q \end{pmatrix}$

6.3 PROOF OF THEOREM 4

We start with an auxiliary result that asserts that a decomposition of the form (48) is characterized by L.

Lemma 4. *Decomposition* (48) *exists if and only if* L *is chosen such that* $P_1 - LP_3$ *is non-singular. In this case,*

$$\operatorname{rank} R = \operatorname{rank}(P_2 - LP_4). \tag{68}$$

Proof. We have:

$$\begin{pmatrix} I_t & \\ L & I_k \end{pmatrix}^{-1} P = \begin{pmatrix} P_4 & P_3 \\ P_2 - LP_4 & P_1 - LP_3 \end{pmatrix} \tag{69}$$

This matrix can be uniquely decomposed as in (47) if and only if $P_1 - LP_3$ is non-singular, and we have the desired value for $\operatorname{rank} R$. □

Now we start the actual proof of Theorem 4. If we assume that decomposition (48) exists for P, then Lemma 4 yields

$$P = \begin{pmatrix} I_t & \\ L & I_k \end{pmatrix} \cdot \begin{pmatrix} P_4 - P_3(P_1 - LP_3)^{-1}(P_2 - LP_4) & P_3 \\ & P_1 - LP_3 \end{pmatrix} \cdot \begin{pmatrix} I_t & \\ (P_1 - LP_3)^{-1}(P_2 - LP_4) & I_k \end{pmatrix} \tag{70}$$

It follows:

- (49) is obvious from (70).
- $\mathbb{K}^k = \operatorname{im}(P_1 - LP_3) \leqslant \operatorname{im} P_1 + \operatorname{im} L$. Thus, $k \leqslant p_1 + \operatorname{rank} L$, which yields (50).
- $\operatorname{im}(P_2 - LP_4) = (P_2 - LP_4)(\mathbb{K}^k) \geqslant (P_2 - LP_4)(\ker P_4) = P_2(\ker P_4)$. (51) now follows from (65) and (68).
- (52) is a direct computation:

$$\begin{aligned} p_2 &= \operatorname{rank}(P_2 - LP_4 + LP_4) \\ &\leqslant \operatorname{rank}(P_2 - LP_4) + \operatorname{rank}(LP_4) \\ &\leqslant \operatorname{rank} R + \operatorname{rank} L. \end{aligned}$$

6.4 PROOF OF THEOREM 5, CASE $p_2 \leqslant t+k-p_1-p_4$

In this section, we provide an algorithm to construct an appropriate decomposition, in the case where $p_2 \leqslant t+k-p_1-p_4$ (Figure 31 left). This means that, using Lemma 4, we have to build a matrix L that satisfies

$$\begin{cases} P_1 - LP_3 \text{ is non-singular,} \\ \operatorname{rank} L = k - p_1, \\ \operatorname{rank}(P_2 - LP_4) = t - p_4. \end{cases}$$

6.4.1 *Sufficient conditions*

We first derive a set of sufficient conditions that ensure that L satisfies the two following properties: $P_1 - LP_3$ non singular (Lemma 5) and $\operatorname{rank}(P_2 - LP_4) = t - p_4$ (Lemma 6).

Lemma 5. *If* $\operatorname{rank} L = k - p_1$ *and* $\operatorname{im} P_1 \oplus LP_3(\ker P_1) = \mathbb{K}^k$, *then* $P_1 - LP_3$ *is non-singular.*

Proof. We denote with $\mathcal{U}$ a complement of $\ker P_1$ in $\mathbb{K}^k$, i.e., $\mathbb{K}^k = \ker P_1 \oplus \mathcal{U}$. This implies $\operatorname{im} P_1 = P_1(\mathbb{K}^k) = P_1(\mathcal{U})$. Now, let $\operatorname{im} P_1 \oplus LP_3(\ker P_1) = \mathbb{K}^k$. Hence, $\dim LP_3(\ker P_1) = k - p_1 = \operatorname{rank} L$ from which we get $\operatorname{im} L = LP_3(\ker P_1)$. In particular, $LP_3(\mathcal{U}) \leqslant \operatorname{im} L = LP_3(\ker P_1)$. Further,

$$\begin{aligned} \operatorname{im}(P_1 - LP_3) &= (P_1 - LP_3)(\mathcal{U} \oplus \ker P_1) \\ &= (P_1 - LP_3)(\mathcal{U}) + LP_3(\ker P_1) \end{aligned}$$

As $LP_3(\mathcal{U}) \leqslant LP_3(\ker P_1)$ and $\operatorname{im} P_1 \cap LP_3(\ker P_1) = \{0\}$, we have:

$$\begin{aligned} \operatorname{im}(P_1 - LP_3) &= P_1(\mathcal{U}) + LP_3(\ker P_1) \\ &= \operatorname{im} P_1 + LP_3(\ker P_1) \\ &= \mathbb{K}^k \end{aligned}$$

as desired. □

Lemma 6. *If, for every vector v of* $\operatorname{im} P_4$, L *satisfies* $Lv \in P_2 P_4^{-1}(\{v\})$, *then* $\operatorname{rank}(P_2 - LP_4) = t - p_4$.

Proof. In the proof of Theorem 4 (Section 6.3) we already showed that $\operatorname{im}(P_2 - LP_4) \geqslant P_2(\ker P_4)$.

Let now $Lv \in P_2 P_4^{-1}(\{v\})$ for all $v \in \operatorname{im} P_4$. If $u \in \mathbb{K}^t$, we have:

$$\begin{aligned} (P_2 - LP_4)u &= P_2 u - LP_4 u \\ &\in P_2 u - P_2 P_4^{-1}(\{P_4 u\}) \\ &\in P_2 u - P_2(u + \ker P_4) \\ &\in P_2(\ker P_4) \end{aligned}$$

Therefore, $\operatorname{im}(P_2 - LP_4) = P_2(\ker P_4)$. Thus, $\operatorname{rank} P_2 - LP_4 = t - p_4$. □

The following lemma summarizes the two previous results:

Lemma 7. *Let $\mathcal{Y}$ be a complement of* $\operatorname{im} P_1$ *and $\mathcal{T}$ a complement of* $P_4(\ker P_2)$ *in* $\operatorname{im} P_4$. *If*

$$\begin{cases} \operatorname{im} L = \mathcal{Y}, \\ L \cdot P_3(\ker P_1) = \mathcal{Y}, \\ L \cdot v \in P_2 P_4^{-1}(\{v\}), \forall v \in \mathcal{T}, \\ L \cdot P_4(\ker P_2) = \{0\}, \end{cases}$$

then L *is an optimal solution*[3].

6.4.2 *Building* L

We now build a matrix L that satisfies the previous set of sufficient conditions. For all v in a complement $\mathcal{T}$ of $P_4(\ker P_2)$, L has to satisfy $Lv \in P_2 P_4^{-1}(\{v\})$. We first show that, given a suitable domain and image, it is possible to build a bijective linear mapping that satisfies this property.

Lemma 8. *Let* $P_4(\ker P_2) \oplus \mathcal{T} = \operatorname{im} P_4$ *and* $P_2(\ker P_4) \oplus \mathcal{V} = \operatorname{im} P_2$. *If we define the subspace*

$$\mathcal{F} = P_4^{-1}(\mathcal{T}) \cap P_2^{-1}(\mathcal{V}),$$

then the mapping $f : \mathcal{T} \to \mathcal{V}$ such that for all $v \in \mathcal{F}$, $f(P_4 v) = P_2 v$, is well defined and is an isomorphism.

3 The proposed set of sufficient conditions is stronger than necessary; if we replace the last condition with $L \cdot P_4(\ker P_2) \leqslant P_2(\ker P_4)$, if and only if holds.

Proof. We prove the lemma by first considering two functions f_1 and f_2 that are P_4 and P_2 restricted to $\mathcal{F}$ as shown in the diagram. We show that both are isomorphisms. Then $f = f_2 \circ f_1^{-1}$ is the desired function.

$$\begin{array}{ccc} & \mathcal{T} & \xrightarrow{\quad f \quad} & \mathcal{V} & \\ f_1 : x \mapsto P_4 x & \nwarrow & & \nearrow & f_2 : x \mapsto P_2 x \\ & & \mathcal{F} & & \end{array}$$

We begin with the surjectivity of f_1. Let $x \in \mathcal{T}$. As $\mathcal{T} \leqslant \operatorname{im} P_4$, there exists a vector v such that $P_4 v = x$. The coset $v + \ker P_4$ is obviously a subset of $P_4^{-1}(\mathcal{T})$. Additionally, its image under P_2, the coset $P_2(v + \ker P_4) = P_2 v + P_2 \ker P_4$ contains a unique representative $P_2 v_f$ in $\mathcal{V}$, since $\operatorname{im} P_2 = P_2(\ker P_4) \oplus \mathcal{V}$. Therefore, $v_f \in P_2^{-1}(\mathcal{V})$, and thus $v_f \in \mathcal{F}$ and $f_1(v_f) = x$, as desired.

We now prove that f_1 is injective. Let $v \in \ker f_1 \leqslant \mathcal{F}$. We have $P_2 v \in \mathcal{V}$. Since $v \in \ker P_4$, $P_2 v \in P_2 \ker P_4$. Since $P_2(\ker P_4) \cap \mathcal{V} = \{0\}$, $P_2 v = 0$ and thus $v \in \ker P_2$. Equation (57) shows that $v = 0$, as desired. Thus, f_1 is bijective.

The proof that $f_2 : \mathcal{F} \to \mathcal{V}$, $v \mapsto P_2 v$ is bijective is analogous. It follows that $f = f_2 \circ f_1^{-1}$ is the desired isomorphism. □

As explained below, we now build a matrix L that matches the conditions listed in Lemma 7. As they involve two spaces that may not be in a direct sum, $P_3(\ker P_1)$ and a complement of $P_4(\ker P_2)$ in $\operatorname{im} P_4$, some precautions must be taken.

We first construct the image $\mathcal{Y}$ of L. It must be a complement of $\operatorname{im} P_1$ and must contain a complement $\mathcal{Y}_1$ of $P_2(\ker P_4)$ in $\operatorname{im} P_2$. From $p_2 \leqslant t + k - p_1 - p_4$ we get $t - p_4 \geqslant p_1 + p_2 - k$ and thus $\dim(P_2(\ker P_4)) \geqslant \dim(\operatorname{im} P_1 \cap \operatorname{im} P_2)$ using (65) and (66). Therefore, we can use the Lemma 3 to build a space $\mathcal{Y}_1$ such that:

$$\begin{cases} \mathcal{Y}_1 \oplus P_2(\ker P_4) = \operatorname{im} P_2, \\ \mathcal{Y}_1 \cap \operatorname{im} P_1 \cap \operatorname{im} P_2 = \{0\}. \end{cases}$$

We then complete $\mathcal{Y}_1$ to form a complement $\mathcal{Y}$ of $\operatorname{im} P_1$.

We now decompose $\mathbb{K}^t$ the following way:

$$\underbrace{\mathcal{X}_1}_{P_3(\ker P_1)} \oplus \overbrace{\mathcal{X}_2 \oplus \mathcal{X}_3 \oplus P_4(\ker P_2)}^{\operatorname{im} P_4} \oplus \mathcal{X}_4 = \mathbb{K}^t$$

We define $\mathcal{X}_2 = P_3(\ker P_1) \cap \operatorname{im} P_4$. $\mathcal{X}_2 \cap P_4(\ker P_2) = \{0\}$ according to equation (61). Then, we define $\mathcal{X}_3$ as a complement of $P_4(\ker P_2) \oplus \mathcal{X}_2$ in $\operatorname{im} P_4$ and $\mathcal{X}_1$ as a complement of $\mathcal{X}_2$ in $P_3(\ker P_1)$. $\mathcal{X}_4$ is defined as a complement of $\mathcal{X}_1 \oplus \mathcal{X}_2 \oplus \mathcal{X}_3 \oplus P_4(\ker P_2)$.

Finally, we build L through the associated mapping, itself defined using a direct sum of linear mappings defined on the following subspaces of $\mathbb{K}^t$:

- We use Lemma 8 to construct a bijective linear mapping f from $\mathcal{T} = \mathcal{X}_2 \oplus \mathcal{X}_3$ onto $\mathcal{V} = \mathcal{Y}_1$. By definition, for all $v \in \mathcal{T}$, f satisfies $f(v) \in P_2 P_4^{-1}(\{v\})$. Furthermore, as f is bijective, its restriction on $\mathcal{X}_2$ is itself bijective onto $f(\mathcal{X}_2)$.

- We complete this bijective linear mapping with g, a bijective linear mapping between $\mathcal{X}_1$ and a complement $\mathcal{Y}_2$ of $f(\mathcal{X}_2)$ in $\mathcal{Y}$. Such a mapping exists as we have $\dim \mathcal{X}_1 - \dim \mathcal{Y}_2 = \dim \mathcal{X}_1 + \dim \mathcal{X}_2 - \dim \mathcal{Y} = \dim(P_3(\ker P_1)) - (k - p_1) = 0$. This way, the restriction of $f \oplus g$ on $\mathcal{X}_1 \oplus \mathcal{X}_2 = P_3(\ker P_1)$ is bijective onto $\mathcal{Y}$.
- We consider the mapping h that maps $P_4(\ker P_2) \oplus \mathcal{X}_4$ to $\{0\}$.

$$\begin{array}{ccccc} \mathcal{X}_1 & & \mathcal{X}_2 \oplus \mathcal{X}_3 & & P_4(\ker P_2) \oplus \mathcal{X}_4 \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle h} \\ \mathcal{Y}_2 & & \mathcal{Y}_1 & & \{0\} \end{array}$$

The matrix associated with the linear mapping $f \oplus g \oplus h$ satisfies all the conditions of Lemma 7, and is therefore an optimal solution.

This method is summarized in Algorithm 3, which allows us to construct a solution for Theorem 5. Its key part is the construction of a basis of $\mathcal{F}$, which uses a generalized pseudo-inverse $P_4^\dagger$ (resp. $P_2^\dagger$) of P_4 (resp. P_2), i.e., a matrix verifying $P_4 P_4^\dagger P_4 = P_4$ (resp. $P_2 P_2^\dagger P_2 = P_2$). This algorithm is a main contribution of this article.

Algorithm 3 Constructing L (Theorem 5), case $p_2 \leqslant t + k - p_1 - p_4$

Require: t,k and $P \in GL_{t+k}(\mathbb{K})$ such that $p_2 \leqslant t + k - p_1 - p_4$
Ensure: L

$Y_1 \leftarrow$ Alg. 2 with $A = P_2 \cdot \overline{\ker} P_4$, $B = P_1 \mathbin{\overline{\cap}} P_2$, $C = P_2$
$Y \leftarrow \begin{pmatrix} Y_1 & I_t \mathbin{\overline{\ominus}} \begin{pmatrix} Y_1 & P_1 \end{pmatrix} \end{pmatrix}$
$X_2 \leftarrow (P_3 \cdot \overline{\ker} P_1) \mathbin{\overline{\cap}} P_4$
$X_3 \leftarrow P_4 \mathbin{\overline{\ominus}} \begin{pmatrix} (P_4 \cdot \ker P_2) & X_2 \end{pmatrix})$
$X_1 \leftarrow (P_3 \cdot \overline{\ker} P_1) \mathbin{\overline{\ominus}} X_2$
$X_4 \leftarrow I_t \mathbin{\overline{\ominus}} \begin{pmatrix} P_4 & P_3 \cdot \overline{\ker} P_1 \end{pmatrix}$
$F \leftarrow \begin{pmatrix} \overline{\ker} P_4 & P_4^\dagger \cdot \begin{pmatrix} X_2 & X_3 \end{pmatrix} \end{pmatrix} \mathbin{\overline{\cap}} \begin{pmatrix} \overline{\ker} P_2 & P_2^\dagger \cdot Y_1 \end{pmatrix}$
$Y_2 \leftarrow Y \mathbin{\overline{\ominus}} (P_2 \cdot (\begin{pmatrix} \overline{\ker} P_4 & P_4^\dagger \cdot X_2 \end{pmatrix} \mathbin{\overline{\cap}} F))$
$L_R \leftarrow \begin{pmatrix} P_4 \cdot F & X_1 & P_4 \cdot \overline{\ker} P_2 & X_4 \end{pmatrix}$
$L_L \leftarrow \begin{pmatrix} P_2 \cdot F & Y_2 & Z \end{pmatrix}$, where Z is a zero filled matrix such that L_L has the same number of columns as L_R
return $L_L \cdot L_R^{-1}$

Inspection of this algorithm shows that its arithmetic cost is $O((t+k)^3)$.

6.4.3 *Example*

We illustrate our algorithm with a concrete example. Motivated by our main application (Theorem 2) we choose as base field $\mathbb{K} = \mathbb{F}_2$. For $t = 4$ and $k = 3$, we consider the matrix

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} = \left(\begin{array}{cccc|ccc} 1 & 1 & . & . & 1 & . & 1 \\ . & . & 1 & . & . & 1 & . \\ 1 & 1 & . & 1 & 1 & 1 & 1 \\ . & . & . & . & . & 1 & 1 \\ \hline 1 & . & 1 & 1 & . & . & . \\ 1 & 1 & . & . & 1 & 1 & 1 \\ . & 1 & . & . & . & . & . \end{array}\right).$$

We observe $p_2 = 3 \leqslant t+k-p_1-p_4 = 4+3-1-3$. Therefore, we can use Algorithm 3 to compute a suitable L.

The first step is to compute Y_1. We have:

$$P_2 \cdot \overline{\ker} P_4 = \begin{pmatrix} 1 \\ . \\ 1 \end{pmatrix} \text{ and } P_1 \overline{\cap} P_2 = \begin{pmatrix} . \\ 1 \\ . \end{pmatrix}.$$

Using Algorithm 2, we get

$$Y_1 = \begin{pmatrix} 1 & 1 \\ 1 & . \\ 1 & . \end{pmatrix}.$$

Then, we complete it to form a complement of $\overline{\mathrm{im}}\, P_1$:

$$Y = \begin{pmatrix} 1 & . \\ . & 1 \\ . & 1 \end{pmatrix}.$$

The next step computes the different domains:

$$X_2 = \begin{pmatrix} 1 \\ 1 \\ . \\ . \end{pmatrix}, X_3 = \begin{pmatrix} 1 \\ . \\ . \\ . \end{pmatrix}, X_1 = \begin{pmatrix} . \\ . \\ . \\ 1 \end{pmatrix} \text{ and } X_4 = ().$$

To compute F, we need pseudo-inverses of P_4 and P_2:

$$P_4^\dagger = \begin{pmatrix} . & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ 1 & . & 1 & . \end{pmatrix} \text{ and } P_2^\dagger = \begin{pmatrix} . & 1 & 1 \\ . & . & 1 \\ . & . & . \\ 1 & 1 & 1 \end{pmatrix}.$$

We then obtain:

$$F = \begin{pmatrix} . & . \\ 1 & . \\ . & 1 \\ 1 & . \end{pmatrix}.$$

Then, we compute Y_2:

$$Y_2 = \begin{pmatrix} 1 \\ . \\ . \end{pmatrix}.$$

Now we can compute L. With

$$L_R = \begin{pmatrix} 1 & . & . & . \\ . & 1 & . & 1 \\ . & . & . & 1 \\ . & . & 1 & . \end{pmatrix}, L_L = \begin{pmatrix} 1 & 1 & 1 & . \\ 1 & . & . & . \\ 1 & . & . & . \end{pmatrix}$$

we get

$$L = L_L \cdot L_R^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & . & . & . \\ 1 & . & . & . \end{pmatrix}.$$

The final decomposition is now obtained using (70):

$$P = \left(\begin{array}{cccc|ccc} 1 & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . \\ . & . & . & 1 & . & . & . \\ \hline 1 & 1 & 1 & 1 & 1 & . & . \\ 1 & . & . & . & . & 1 & . \\ 1 & . & . & . & . & . & 1 \end{array}\right) \left(\begin{array}{ccc|ccc} & 1 & . & . & 1 & . & 1 \\ . & . & 1 & . & . & 1 & . \\ . & 1 & . & 1 & 1 & 1 & 1 \\ 1 & . & . & . & . & 1 & 1 \\ \hline . & . & . & . & & 1 & 1 \\ . & . & . & . & . & 1 & . \\ . & . & . & & 1 & . & 1 \end{array}\right) \left(\begin{array}{cccc|ccc} 1 & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . \\ . & . & . & 1 & . & . & . \\ \hline . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . \\ 1 & . & . & . & . & . & 1 \end{array}\right).$$

This decomposition satisfies $\operatorname{rank} L = 2$ and $\operatorname{rank} R = 1$, thus matching the bounds of Theorem 4.

If we consider the application presented in Theorem 2, this decomposition provides a way to implement in hardware the permutation associated with P on 128 elements, arriving in chunks of 8 during 16 cycles. This yields an implementation consisting of a permutation network of 4 2×2-switches, followed by a block of 8 RAM banks, followed by another permutation network with 8 2×2-switches.

6.5 PROOF OF THEOREM 5, CASE $p_2 \geqslant t+k-p_1-p_4$

In this case, the third inequality in Theorem 4 is restrictive (Figure 31 right). Using again Lemma 4, we have to build a matrix L satisfying:

$$\begin{cases} P_1 - LP_3 \text{ is non-singular} \\ \operatorname{rank} L = k - p_1 \\ \operatorname{rank}(P_2 - LP_4) = p_1 + p_2 - k \end{cases}$$

As in the previous section, we will first provide a set of sufficient conditions for L and then build it.

6.5.1 *Sufficient conditions*

The set of conditions that we will derive in this subsection will be slightly more complex than in the previous section, as we cannot reach the intrinsic bound of $\operatorname{rank}(P_2 - LP_4)$. Particularly, we cannot use Lemma 6 directly.

Lemma 9. *If $\mathcal{W}$ is such that $\mathcal{W} \oplus \operatorname{im} P_1 = \mathbb{K}^k$ and $\mathcal{T}$ is such that:*

$$\begin{cases} \mathcal{T} \cap P_4(\ker P_2) = \{0\} \\ \mathcal{T} \leqslant \operatorname{im} P_4 \\ \dim \mathcal{T} = k - p_1, \end{cases}$$

then if L satisfies

$$\begin{cases} \operatorname{im} L = \mathcal{W} \\ L \cdot P_3(\ker P_1) = \mathcal{W} \\ L \cdot v \in P_2 P_4^{-1}(\{v\}), \forall v \in \mathcal{T} \\ L \cdot P_4(\ker P_2) = \{0\} \end{cases}$$

then L is a solution[4] that satisfies $\operatorname{rank} L = k - p_1$ and $\operatorname{rank}(P_2 - LP_4) = p_1 + p_2 - k$.

Proof. Let L be a matrix that satisfies the conditions above. Using Lemma 5 as before, we get that $\operatorname{rank} L = k - p_1$ and $P_1 - LP_3$ invertible.

Now, with the definition of $\mathcal{T}$, we can define a dimension $p_1 + p_2 + p_4 - t - k$ space $\mathcal{T}'$ such that $\operatorname{im} P_4 = P_4(\ker P_2) \oplus \mathcal{T} \oplus \mathcal{T}'$. Then, we define a matrix L' such that:

$$\begin{cases} L' \cdot v \in Lv - P_2 P_4^{-1}(\{v\}), \text{ for all } v \in \mathcal{T}' \\ L' \cdot v = 0, \text{ for all } v \text{ in } P_4(\ker P_2) \oplus \mathcal{T} \\ L' \cdot v = 0, \text{ for all } v \text{ in a complement of } \operatorname{im} P_4, \end{cases}$$

L' is therefore a rank $p_1 + p_2 + p_4 - t - k$ matrix such that for all $v \in \operatorname{im} P_4$, $(L - L')v \in P_2 P_4^{-1}(\{v\})$. We apply Lemma 6 on $L - L'$ and get:

$$\begin{aligned} \operatorname{rank}(P_2 - LP_4) &= \operatorname{rank}(P_2 - (L - L')P_4 - L'P_4) \\ &\leqslant \operatorname{rank}(P_2 - (L - L')P_4) + \operatorname{rank}(L'P_4) \\ &\leqslant \dim P_2(\ker P_4) + \operatorname{rank} L' \\ &\leqslant p_1 + p_2 - k, \end{aligned}$$

4 If we replace the last condition with $L \cdot P_4(\ker P_2) \leqslant P_2(\ker P_4)$, this set of conditions is actually equivalent to having an optimal solution L that satisfies $\operatorname{rank} L = k - p_1$.

as desired. □

6.5.2 *Building* L

We will build a matrix L that matches the conditions listed in Lemma 9. As before, we consider the image $\mathcal{Y}$ of L first. We will design it such that it is a complement of $\operatorname{im} P_1$, and that is contained in a complement $\mathcal{Y}'$ of $P_2(\ker P_4)$ in $\operatorname{im} P_2$. Using $p_2 \geqslant t + k - p_1 - p_4$ with (65) and (66) we get $\dim(P_2 \ker P_4) \leqslant \dim(\operatorname{im} P_1 \cap \operatorname{im} P_2)$. With Lemma 3, we can construct a space $\mathcal{Y}$ that satisfies

$$\begin{cases} \mathcal{Y} \oplus (\operatorname{im} P_1 \cap \operatorname{im} P_2) = \operatorname{im} P_2 \\ \mathcal{Y} \cap P_2(\ker P_4) = \{0\} \end{cases}$$

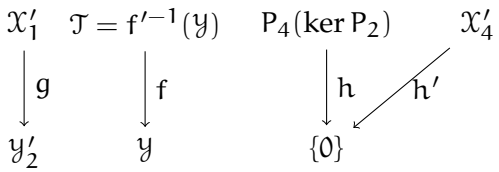
This space satisfies $\mathcal{Y} \oplus \operatorname{im} P_1 = \operatorname{im} P_2 + \operatorname{im} P_1 = \mathbb{K}^k$ according to (58), and can be completed to a complement $\mathcal{Y}'$ of $P_2(\ker P_4)$ in $\operatorname{im} P_2$. Note that we will use $\mathcal{Y}'$ only to define f; the image of L will be $\mathcal{Y}$.

Now, as before, we build L through the associated mapping, itself defined using a direct sum of linear mappings defined on the same subspaces of $\mathbb{K}^t$ as in Section 6.4.2.

- We use Lemma 8 to construct a first bijective linear mapping f' between $\mathcal{T}' = \mathcal{X}_2 \oplus \mathcal{X}_3$ and $\mathcal{Y}'$. As f' is bijective, we can define $\mathcal{T} = f'^{-1}(\mathcal{Y})$ and $f = f'|_{\mathcal{T}}$. Thus, $\mathcal{T}$ satisfies the properties in Lemma 9 and L the condition for all $v \in \mathcal{T}, Lv \in P_2 P_4^{-1}(\{v\})$.
- Then, we consider a complement $\mathcal{X}_1'$ of $\mathcal{T} \cap \mathcal{X}_2$ in $P_3(\ker P_1)$, a complement $\mathcal{Y}_2'$ of $f(\mathcal{T} \cap \mathcal{X}_2)$ in $\mathcal{Y}$ and a bijective linear mapping g between $\mathcal{X}_1'$ and $\mathcal{Y}_2'$. This way, the restriction of $f \oplus g$ on $P_3(\ker P_1)$ is bijective onto $\mathcal{Y}$.

The rest of the algorithm in similar to the previous case:

- We consider the mapping h that maps $P_4(\ker P_2)$ to $\{0\}$.
- To complete the definition of L, we take a mapping h' between a complement $\mathcal{X}_4'$ of $\mathcal{X}_1' \oplus \mathcal{T} \oplus P_4(\ker P_2)$ and $\{0\}$.



The matrix associated with the mapping $f \oplus g \oplus h \oplus h'$ satisfies all the conditions of Lemma 9, and is therefore an optimal solution.

Algorithm 4 summarizes this method, and allows to construct a solution for Theorem 5, in the case where $p_2 > t + k - p_1 - p_4$. This algorithm is a main contribution of this article.

As in the previous case, this algorithm has an arithmetic cost cubic in $t + k$.

Algorithm 4 Constructing L (Theorem 5), case $p_2 > t+k-p_1-p_4$

Require: t,k, $P \in GL_{t+k}(\mathbb{K})$ such that $p_2 > t+k-p_1-p_4$
Ensure: L
$Y \leftarrow$ Alg. 2 with $A = P_1 \overline{\cap} P_2$, $B = P_2 \cdot \overline{\ker} P_4$ and $C = P_2$
$X_2 \leftarrow (P_3 \cdot \overline{\ker} P_1) \overline{\cap} P_4$
$X_3 \leftarrow P_4 \overline{\ominus} \begin{pmatrix} P_4 \cdot \overline{\ker} P_2 & X_2 \end{pmatrix}$
$F \leftarrow \begin{pmatrix} \overline{\ker} P_4 & P_4^\dagger \cdot \begin{pmatrix} X_2 & X_3 \end{pmatrix} \end{pmatrix} \overline{\cap} \begin{pmatrix} \overline{\ker} P_2 & P_2^\dagger \cdot Y \end{pmatrix}$
$X_1' \leftarrow (P_3 \cdot \ker P_1) \overline{\ominus} ((P_4 \cdot F) \overline{\cap} X_2)$
$X_4 \leftarrow I_t \overline{\ominus} \begin{pmatrix} X_1' & P_4 \cdot F & P_4 \cdot \overline{\ker} P_2 \end{pmatrix}$
$Y_2' \leftarrow Y \overline{\ominus} (P_2 \cdot (F \overline{\cap} \begin{pmatrix} P_4^\dagger \cdot X_2 & \overline{\ker} P_4 \end{pmatrix}))$
$L_R \leftarrow \begin{pmatrix} P_4 \cdot F & X_1' & P_4 \cdot \overline{\ker} P_2 & X_4 \end{pmatrix}$
$L_L \leftarrow \begin{pmatrix} P_2 \cdot F & Y_2' & Z \end{pmatrix}$, where Z is a zero filled matrix such that L_L has the same number of columns as L_R
return $L_L \cdot L_R^{-1}$

6.5.3 *Example*

We now consider, for $\mathbb{K} = \mathbb{F}_2$, $t = 4$ and $k = 3$, the matrix

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} = \left(\begin{array}{cccc|ccc} . & 1 & 1 & 1 & 1 & . & . \\ 1 & . & . & 1 & . & 1 & 1 \\ . & 1 & 1 & 1 & . & 1 & 1 \\ 1 & 1 & . & 1 & . & 1 & 1 \\ \hline 1 & . & . & 1 & . & 1 & . \\ . & . & . & 1 & . & 1 & . \\ 1 & . & 1 & 1 & 1 & 1 & . \end{array}\right).$$

We observe $p_2 = 3 > t+k-p_1-p_4 = 4+3-2-3$. Therefore, we use Algorithm 4 to compute a suitable L.

The first step is to compute Y. We have:

$$P_2 \cdot \overline{\ker} P_4 = \begin{pmatrix} . \\ 1 \\ 1 \end{pmatrix} \text{ and } P_1 \overline{\cap} P_2 = \begin{pmatrix} . \\ 1 \\ . \end{pmatrix}.$$

Using Algorithm 2, we get

$$Y_1 = \begin{pmatrix} 1 & 1 \\ 1 & . \\ 1 & . \end{pmatrix}.$$

Then, we complete it to form a complement of $\operatorname{im} P_1$:

$$Y = \begin{pmatrix} 1 & . \\ 1 & . \\ . & 1 \end{pmatrix}.$$

The next step computes the different domains:

$$X_2 = () \text{ and } X_3 = \begin{pmatrix} 1 & . \\ . & 1 \\ 1 & . \\ . & . \end{pmatrix}.$$

To compute F, we need pseudo-inverses of P_4 and P_2:

$$P_4^\dagger = \begin{pmatrix} . & . & . & . \\ . & 1 & . & 1 \\ . & . & 1 & 1 \\ . & 1 & . & . \end{pmatrix} \text{ and } P_2^\dagger = \begin{pmatrix} 1 & 1 & . \\ . & . & . \\ 1 & . & 1 \\ . & 1 & . \end{pmatrix}$$

and get

$$F = \begin{pmatrix} 1 \\ 1 \\ . \\ . \end{pmatrix}.$$

Next we compute the remaining subspaces that depend on F:

$$X_1' = \begin{pmatrix} . \\ 1 \\ 1 \\ 1 \end{pmatrix}, X_4 = \begin{pmatrix} 1 \\ . \\ . \\ . \end{pmatrix}, \text{ and } Y_2' = \begin{pmatrix} 1 \\ . \\ 1 \end{pmatrix}.$$

Noe we can compute L. With

$$L_R = \begin{pmatrix} 1 & . & 1 & 1 \\ 1 & 1 & . & . \\ 1 & 1 & 1 & . \\ . & 1 & 1 & . \end{pmatrix}, L_L = \begin{pmatrix} 1 & 1 & . & . \\ . & . & . & . \\ 1 & 1 & . & . \end{pmatrix}$$

we get

$$L = L_L \cdot L_R^{-1} = \begin{pmatrix} . & 1 & . & . \\ . & . & . & . \\ . & 1 & . & . \end{pmatrix}.$$

The final decomposition is obtained using (70):

$$P = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ \hline . & 1 & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \end{array}\right) \left(\begin{array}{ccc|ccc} . & 1 & . & 1 & 1 & . \\ 1 & . & . & . & . & 1 \\ . & 1 & 1 & . & . & 1 \\ 1 & 1 & . & . & . & 1 \\ \hline . & . & . & . & . & 1 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{array}\right) \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ \hline . & . & 1 & . & 1 & . \\ . & . & . & 1 & . & . \\ . & . & . & . & . & 1 \end{array}\right).$$

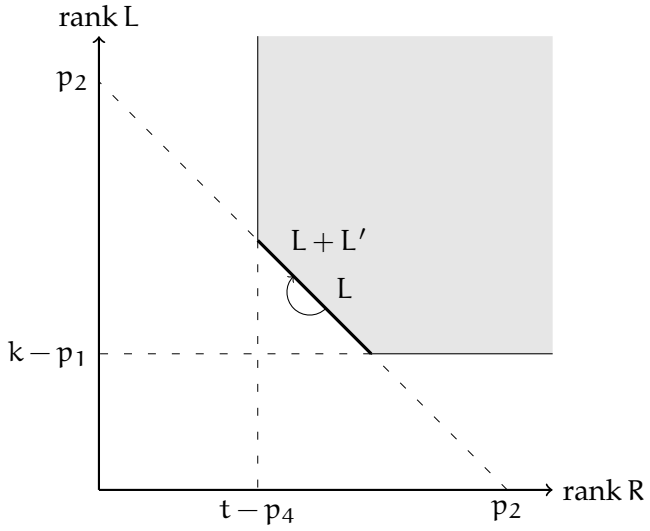


Figure 32: L' trades a rank of L for a rank of R on the associated decomposition.

This decomposition satisfies $\operatorname{rank} L = 1$ and $\operatorname{rank} R = 2$, thus matching the bounds of Theorem 4.

As before, if we consider the application of Theorem 2. The decomposition shows that we can implement in hardware the permutation associated with P on 128 elements, arriving in chunks of 8 during 16 cycles through a permutation network of 8 2×2-switches, followed by a block of 8 RAM banks, followed by another permutation network with 4 2×2-switches.

6.6 RANK EXCHANGE

The solution built in Section 6.5.2 satisfies $\operatorname{rank} L = k - p_1$ and $\operatorname{rank}(P_2 - LP_4) = p_1 + p_2 - k$. In this section, we will show that it is possible to construct a solution for all possible pairs $(\operatorname{rank} L, \operatorname{rank}(P_2 - LP_4))$ matching the bounds in Theorem 4. First, we will construct a rank 1 matrix L' that will trade a rank of L for a rank of $P_2 - LP_4$ (i.e., $\operatorname{rank}(L + L') = 1 + \operatorname{rank} L$, $\operatorname{rank}(P_2 - (L + L')P_4) = \operatorname{rank}(P_2 - LP_4) - 1$ and $P_1 - (L + L')P_3$ is non-singular) (see Figure 32). This method can then be applied several times, until $\operatorname{rank}(P_2 - LP_4)$ reaches its own bound, $t - p_4$.

We assume that L satisfies the following conditions:

$$\begin{cases} P_1 - LP_3 \text{ is non-singular} \\ \operatorname{rank} L + \operatorname{rank}(P_2 - LP_4) = p_2 \\ \operatorname{rank}(P_2 - LP_4) > t - p_4 \end{cases}$$

As in the previous sections, we first formulate sufficient conditions on L', before building it.

6.6.1 *Sufficient conditions*

We now define $C = P_4 - P_3(P_1 - LP_3)^{-1}(P_2 - LP_4)$.

Lemma 10. *If* $z \in \mathbb{K}^t$ *satisfies* $z \notin \ker(P_2 - LP_4) + \ker P_4$ *and* L' *satisfies:*

$$\begin{cases} \operatorname{rank} L' = 1 \\ L'P_4 \ker(P_2 - LP_4) = \{0\} \\ L'P_4 z = (P_2 - LP_4)z \\ L'Cz \neq 0 \end{cases}$$

Then $L + L'$ *is an optimal solution to our problem that satisfies* $\operatorname{rank}(P_2 - (L + L')P_4) = \operatorname{rank}(P_2 - LP_4) - 1$.

Proof. We first prove that $P_1 - (L + L')P_3 = (I - L'P_3(P_1 - LP_3)^{-1})(P_1 - LP_3)$ is non singular. Let $x \in \ker(I - L'P_3(P_1 - LP_3)^{-1})$. x satisfies:

$$x - L'P_3(P_1 - LP_3)^{-1}x = 0.$$

Therefore, $x \in \operatorname{im} L'$. As $\operatorname{rank} L' = 1$, $\exists \lambda \in \mathbb{K}$, $x = \lambda(P_2 - LP_4)z = \lambda L'P_4 z$. It comes that

$$\lambda L'P_4 z - \lambda L'P_3(P_1 - LP_3)^{-1}(P_2 - LP_4)z = 0.$$

Finally, $\lambda L'Cz = 0$, which implies, as $L'Cz \neq 0$, $\lambda = 0$, as desired.

We now prove that $\operatorname{rank}(P_2 - (L + L')P_4) = \operatorname{rank}(P_2 - LP_4) - 1$. We have already $\ker(P_2 - (L + L')P_4) \leqslant \ker(P_2 - LP_4)$ as $L'P_4 \ker(P_2 - LP_4) = \{0\}$. We also have $(P_2 - (L' + L)P_4)z = 0$. As $z \notin \ker(P_2 - LP_4) + \ker P_4$, $\ker(P_2 - (L + L')P_4) \geqslant \ker(P_2 - LP_4) \oplus \langle z \rangle$. Therefore,

$$\dim \ker(P_2 - (L + L')P_4) \geqslant 1 + \dim \ker(P_2 - LP_4),$$

as desired. □

6.6.2 *Building* L'

Lemma 11. $\ker(P_2 - LP_4) \cap \ker P_4 = \{0\}$

Proof. This is a consequence of (69): the block column $\begin{pmatrix} P_4 \\ P_2 - LP_4 \end{pmatrix}$ has full rank. □

Thus, we have $\dim(\ker(P_2 - LP_4) \oplus \ker P_4) = 2t - p_4 - \operatorname{rank}(P_2 - LP_4) < t$. Decomposition (70) shows that C is non-singular, and using Lemma 11, we have $\dim C^{-1}P_4 \ker(P_2 - LP_4) = \dim P_4 \ker(P_2 - LP_4) = \dim \ker(P_2 - LP_4) = t - \operatorname{rank}(P_2 - LP_4)$. Using Lemma 3, we can build a space $\mathcal{Z}$ such that:

$$\begin{cases} \mathcal{Z} \oplus \ker(P_2 - LP_4) \oplus \ker P_4 = \mathbb{K}^t \\ \mathcal{Z} \cap C^{-1}P_4 \ker(P_2 - LP_4) = \{0\} \end{cases}$$

We can now pick a nonzero element $z \in \mathcal{Z}$ and build a corresponding L':

- If $Cz \in P_4 \ker(P_2 - LP_4) \oplus \langle P_4 z \rangle$: We take a complement $\mathcal{A}$ of $P_4 \ker(P_2 - LP_4) \oplus \langle P_4 z \rangle$ and build L' such that:

  $$\begin{cases} L'P_4 z = (P_2 - LP_4)z \\ L'(P_4 \ker(P_2 - LP_4) \oplus \mathcal{A}) = \{0\} \end{cases}$$

  We have $L'Cz \neq 0$. In fact, Cz can be uniquely decomposed in the form $k + \lambda P_4 z$, where $k \in P_4 \ker(P_2 - LP_4)$ and $\lambda \in \mathbb{K}$. As $z \notin C^{-1}P_4 \ker(P_2 - LP_4)$, $\lambda \neq 0$. Then, $L'Cz = L'k + L'\lambda P_4 z = 0 + \lambda(P_2 - LP_4)z \neq 0$.

- If $Cz \notin P_4 \ker(P_2 - LP_4) \oplus \langle P_4 z \rangle$: The vector $a = Cz - P_4 z$ is outside of $P_4 \ker(P_2 - LP_4) \oplus \langle P_4 z \rangle$. Therefore, it is possible to build a complement $\mathcal{A}$ of $P_4 \ker(P_2 - LP_4) \oplus \langle P_4 z \rangle$ that contains a. Then, we build L' as before:

$$\begin{cases} L'P_4 z = (P_2 - LP_4)z \\ L'(P_4 \ker(P_2 - LP_4) \oplus \mathcal{A}) = \{0\} \end{cases}$$

  As in the previous case, we have $L'Cz = L'a + L'P_4 z = 0 + (P_2 - LP_4)z \neq 0$.

In both cases, the matrix L' we built satisfies the conditions of Lemma 10. Therefore, $L + L'$ is the desired solution.

Algorithm 5 summarizes this method, and allows to build a new optimal solution from a pre-existing one, with a different trade-off. This algorithm is a main contribution of this article.

Algorithm 5 Exchanging ranks between L and R (Theorem 6)

Require: P and a solution L such that $\operatorname{rank}(P_2 - LP_4) > t - p_4$
Ensure: A new optimal solution L with a rank incremented by 1
  $K \leftarrow \overline{\ker}(P_2 - LP_4)$
  $C \leftarrow P_4 - P_3(P_1 - LP_3)^{-1}(P_2 - LP_4)$
  $Z \leftarrow$ Alg. 2 with $A = \begin{pmatrix} K & \overline{\ker} P_4 \end{pmatrix}$, $B = C^{-1}P_4K$ and $C = I_t$
  $z \leftarrow$ first column of Z
  if $Cz \in \begin{pmatrix} P_4 \cdot K & P_4 z \end{pmatrix}$ **then**
    $A \leftarrow I_t \mathbin{\overline{\ominus}} P_4 \cdot \begin{pmatrix} K & z \end{pmatrix}$
  else
    $a \leftarrow (C - P_4)z$
    $A \leftarrow \begin{pmatrix} I_t \mathbin{\overline{\ominus}} \begin{pmatrix} P_4K & P_4z & a \end{pmatrix} & a \end{pmatrix}$
  end if
  $L'_R \leftarrow \begin{pmatrix} P_4 z & P_4 K & A \end{pmatrix}$
  $L'_L \leftarrow \begin{pmatrix} (P_2 - LP_4)z & F \end{pmatrix}$, where F is a zero filled matrix such that L'_L has the same number of columns as L'_R
  return $L + L'_L \cdot L'^{-1}_R$

6.6.3 *Example*

To illustrate Algorithm 5, we continue the example of Section 6.5.3, and the matrix L that we found. We have:

$$K = \begin{pmatrix} 1 & . \\ . & 1 \\ . & . \\ . & . \end{pmatrix} \text{ and } C = \begin{pmatrix} . & 1 & . & 1 \\ 1 & . & . & . \\ . & 1 & 1 & . \\ 1 & 1 & . & . \end{pmatrix}.$$

Using Algorithm 2, we get

$$Z = z = \begin{pmatrix} . \\ . \\ 1 \\ . \end{pmatrix}.$$

As $Cz = \begin{pmatrix} . \\ . \\ 1 \\ . \end{pmatrix}$ is not included in $\begin{pmatrix} P_4K & P_4z \end{pmatrix} = \begin{pmatrix} 1 & . & . \\ . & 1 & . \\ 1 & . & . \\ . & . & 1 \end{pmatrix}$, we compute A as a complement of $\begin{pmatrix} P_4K & P_4z \end{pmatrix}$ that contains $a = \begin{pmatrix} 1 \\ . \\ . \\ . \end{pmatrix}$:

$$\mathcal{A} = \left\langle \begin{pmatrix} 1 \\ . \\ . \\ . \end{pmatrix} \right\rangle.$$

Now, we compute L', using:

$$L'_R = \begin{pmatrix} 1 & . & 1 & 1 \\ . & 1 & . & . \\ 1 & . & 1 & . \\ . & 1 & 1 & . \end{pmatrix}, L'_L = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ 1 & . & . & . \end{pmatrix},$$

and

$$L' = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & 1 & 1 & 1 \end{pmatrix}.$$

Finally, we get the new decomposition, using, as usual, Equation (70):

$$P = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . \\ . & . & . & 1 & . & . & . \\ \hline . & 1 & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . \\ . & . & 1 & 1. & . & . & 1 \end{array}\right) \left(\begin{array}{ccc|ccc} . & 1 & 1 & 1 & 1 & . & . \\ 1 & . & . & . & . & 1 & 1 \\ . & 1 & 1 & . & . & 1 & 1 \\ 1 & 1 & . & . & . & 1 & 1 \\ \hline . & . & . & . & . & . & 1 \\ . & . & . & . & . & 1 & . \\ . & . & . & . & 1 & 1 & . \end{array}\right) \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . \\ . & . & . & 1 & . & . & . \\ \hline . & . & . & . & 1 & . & . \\ . & . & . & 1 & . & 1 & . \\ . & . & . & . & . & . & 1 \end{array}\right).$$

As expected, the left off-diagonal rank has increased by one, while the right one has decreased by one. The two different decompositions that we now have for P cover all the possible tradeoffs that minimize the off-diagonal ranks.

6.7 CONCLUSION

The problem of implementing streaming linear permutations in Chapter 2 has led us to introduce a novel block matrix decomposition that generalizes the classical block-LU factorization, and decomposes a 2×2-blocked invertible matrix into a product

of three matrices: lower block unitriangular, upper block triangular, and lower block unitriangular matrix, such that the sum of the off-diagonal ranks are minimal. We provided an algorithm that computes an optimal solution with an asympotic number of operations cubic in the matrix size. We note that this decomposition works beyond $\mathbb{F}_2$, and we implemented the algorithm for other finite fields, for rational numbers, for Gaussian rational numbers and for exact real arithmetic for validation. For a floating point implementation however, numerical issues may arise.

7

CHARACTERIZING AND ENUMERATING WALSH-HADAMARD TRANSFORM ALGORITHMS

In this chapter, available in [70], we provide a proof of Theorem 3 that enumerates the algorithms for computing a WHT that consist of a network of stages of butterflies $(I_{2^{n-1}} \otimes H_2)$ interleaved by linear permutations.

7.1 NOTATIONS

We introduce the following notations in this chapter to improve the readability of the different proofs.

Sequence of linear permutations. We consider a sequence P of $n+1$ invertible $n \times n$ bit-matrices

$$P = (P_0, P_1, \ldots, P_n),$$

and the computation, i.e., associated butterfly network

$$\begin{aligned} W(P) =& \pi(P_0) \cdot (I_{2^{n-1}} \otimes \mathrm{DFT}_2) \cdot \pi(P_1) \cdot (I_{2^{n-1}} \otimes \mathrm{DFT}_2) \ldots \\ & \pi(P_{n-1}) \cdot (I_{2^{n-1}} \otimes \mathrm{DFT}_2) \cdot \pi(P_n). \end{aligned}$$

Note that we do not assume a priori that P is such that $W(P) = H_{2^n}$. In fact, we denote this subset, the set of DFT_2*-based linear fast WHT algorithms* with $\mathcal{P}$:

$$\mathcal{P} = \{P = (P_0, P_1, \ldots, P_n) \text{ with } P_i \in \mathrm{GL}_n(\mathbb{F}_2) \mid W(P) = H_{2^n}\}.$$

Product of matrices. The product of the matrices $P_i, P_{i+1}, \cdots, P_{j-1}, P_j$ (a subsequence of P) appears multiple times in this chapter, and we denote it therefore with $P_{i:j}$:

$$P_{i:j} = \prod_{k=i}^{j} P_k.$$

For convenience, we extend this notation by defining $P_{i:j} = I_n$ if $j < i$.

Spreading matrix. Similarly, the following matrix X is recurring:

$$X = \begin{pmatrix} P_{0:n-1}1_b & P_{0:n-2}1_b & \cdots & P_{0:1}1_b & P_0 1_b \end{pmatrix}. \tag{71}$$

Here, $1_b = \begin{pmatrix} 0 & \ldots & 0 & 1 \end{pmatrix}^T$. Thus, X is obtained by concatenating the rightmost columns of the matrices $P_{0:n-1}, \ldots, P_0$. We will refer to this matrix as the *spreading matrix*, as we will see that its invertibility is a necessary and sufficient condition for $W(P)$ to have no zero elements[1].

1 It can be shown that a row of $W(P)$ contains at most $2^{\operatorname{rank} X}$ non-zero elements.

7.2 CHARACTERIZATION OF WHT ALGORITHMS

A naïve approach to check if $P \in \mathcal{P}$ would compute $W(P)$ and compare it against H_{2^n}. Therefore, it would perform $2n+1$ multiplications of $2^n \times 2^n$ matrices, and would have a complexity in $O(n \cdot 2^{3n})$ arithmetic operations. Our objective is to derive an equivalent set of conditions that can be checked with a polynomial complexity.

Lemma 12 provides a necessary and sufficient set of conditions on a sequence of linear permutations such that the corresponding algorithm computes a WHT. A proof of this lemma is given in Section 7.4.

Lemma 12. *$P \in \mathcal{P}$ if and only if the following conditions are satisfied:*

- *The product of the matrices satisfies*

$$P_{0:n} = XX^T. \tag{72}$$

- *The rows of the inverse of the spreading matrix are the last rows of the matrices $P_0^{-1}, \ldots, P_{0:n-1}^{-1}$:*

$$X^{-1} = \begin{pmatrix} P_{0:n-1}^{-T} 1_b & P_{0:n-2}^{-T} 1_b & \cdots & P_{0:1}^{-T} 1_b & P_0^{-T} 1_b \end{pmatrix}^T. \tag{73}$$

This set of conditions is minimal: there are counterexamples that do not satisfy one condition, while satisfying the other.

Cost. With this set of conditions, checking if a given sequence P corresponds to a WHT requires $O(n^4)$ arithmetic operations.

7.3 OTHER TRANSFORMS

We consider here linearly permuted fast algorithms that compute the unscaled and *naturally ordered* WHT. We briefly discuss other WHT variants.

Walsh transform. The *sequency ordered* version, represented by Walsh matrix, only differs by a *bit-reversal* permutation of its outputs. All the results obtained here can be used for the Walsh matrix, after multiplying P_0 by J_n on the left. Particularly, the number of DFT_2-based linear fast Walsh transforms is the same as for the WHT.

Orthogonal WHT. The WHT can be made *orthogonal* by scaling it by a factor of $2^{-n/2}$. Algorithms performing this transform can be obtained with our technique by using the orthogonal DFT_2, i.e. each butterfly is scaled by a factor of $1/\sqrt{2}$.

7.4 PROOF OF LEMMA 12

In this section, we provide a proof of Lemma 12. The main idea of this proof is in deriving a general expression of $W(P)$, assuming only that $W(P)$ has its first row and its first column filled with 1s (Lemma 16). Then, we match this expression with the definition of the WHT to derive necessary and sufficient conditions for an algorithm to compute a WHT. Before that, in Lemma 13, we derive some consequences of the invertibility of the spreading matrix X, particularly on the non-zero elements of $W(P)$. In Lemma 14, we provide a necessary and sufficient condition for $W(P)$ to have a 1-filled first row and column. We begin by defining concepts that will be used throughout this section.

Stages of an algorithm. We will refer to the stage k as an array of DFT_2 composed with the linear permutation associated with P_k: $(I_{2^{n-1}} \otimes \mathrm{DFT}_2)\pi(P_k)$. We call the rightmost stage of an algorithm the stage n, and stage k is to the left of stage $k+1$. We denote with $W_k(P)$ the matrix corresponding to the output on the left of stage k, i.e.,

$$W_k(P) = (I_{2^{n-1}} \otimes \mathrm{DFT}_2)\pi(P_k)\dots(I_{2^{n-1}} \otimes \mathrm{DFT}_2)\pi(P_n).$$

As a consequence, $W(P) = \pi(P_0)W_1(P)$. For practical reasons, we extend this definition for $k = n+1$ by considering that $W_{n+1}(P) = I_n$.

Outputs depending on input i on the left of stage k. For $0 \leqslant i < 2^n$, we denote with $\mathcal{D}_k(i)$ the set of the outputs of the k^{th} stage of the algorithm that depend on the i^{th} input:

$$\mathcal{D}_k(i) = \{j_b \mid W_k(P)[j,i] \neq 0\} \subseteq \mathbb{F}_2^n.$$

The dependency of the whole algorithm on the input i is denoted with $\mathcal{D}(i)$:

$$\mathcal{D}(i) = \{j_b \mid W(P)[j,i] \neq 0\}.$$

Similarly, we denote with $\mathcal{D}_k^+(i)$ (resp. $\mathcal{D}_k^-(i)$) the set of the indices of the outputs for which

$$\mathcal{D}_k^+(i) = \{j_b \mid W_k(P)[j,i] = 1\}, \text{ and } \mathcal{D}_k^-(i) = \{j_b \mid W_k(P)[j,i] = -1\}.$$

7.4.1 *Invertibility of the spreading matrix*

In the following lemma, we justify the name "spreading matrix" that we use for X, by showing that its invertibility conditions the "spread" of non-zero elements through the rows of $W(P)$, and provide some other consequences we will use later.

Lemma 13. *All the outputs of the algorithm depend on the first input, i.e. $\mathcal{D}(0) = \mathbb{F}_2^n$ if and only if the spreading matrix X is invertible. In this case, for every $0 \leqslant i < 2^n$ and $1 \leqslant k \leqslant n$:*

- *The set of dependency at the k^{th} stage on the input i is*

$$\mathcal{D}_k(i) = (P_k\mathcal{D}_{k+1}(i)) \cup (P_k\mathcal{D}_{k+1}(i) + 1_b). \tag{74}$$

- *The non-zero elements of $W_k(P)$ are either 1 or -1:*

$$\mathcal{D}_k(i) = \mathcal{D}_k^+(i) \cup \mathcal{D}_k^-(i). \tag{75}$$

- *The k^{th} stage modifies the set of dependencies such that:*

$$\begin{aligned}\mathcal{D}_k^+(i) =&(P_k\mathcal{D}_{k+1}^+(i) + 1_b) \cup \{j_b \in P_k\mathcal{D}_{k+1}^+(i) \mid j_b^T 1_b = 0\} \cup \\ &\{j_b \in P_k\mathcal{D}_{k+1}^-(i) \mid j_b^T 1_b = 1\}.\end{aligned} \tag{76}$$

Proof. First, we assume that $\mathcal{D}(0) = \mathbb{F}_2^n$, and show that X is invertible. For a given j, the two outputs j_b and $j_b + 1_b$ of a DFT_2 may have a dependency on the first input only if at least one of the two signals j_b and $j_b + 1_b$ that arrive on this DFT_2 depends on that input. Therefore, we have

$$\mathcal{D}_k(0) \subseteq P_k\mathcal{D}_{k+1}(0) \cup (P_k\mathcal{D}_{k+1}(0) + 1_b).$$

We now prove by induction that

$$\mathcal{D}_k(0) \subseteq \langle 1_b, P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle.$$

We already have that $\mathcal{D}_{n+1}(0) = \{0_b\}$. Assuming that the result holds at rank $k+1$, we have:

$$\begin{aligned}
\mathcal{D}_k(0) &\subseteq P_k \mathcal{D}_{k+1}(0) \cup (P_k \mathcal{D}_{k+1}(0) + 1_b) \\
&\subseteq P_k \langle 1_b, P_{k+1} 1_b, \ldots, P_{k+1:n-1} 1_b \rangle \cup \\
&\quad (P_k \langle 1_b, P_{k+1} 1_b, \ldots, P_{k+1:n-1} 1_b \rangle + 1_b) \\
&= \langle P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle \cup \\
&\quad (\langle P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle + 1_b) \\
&= \langle 1_b, P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle.
\end{aligned}$$

Which yields the result. As a consequence, we have

$$\mathcal{D}(0) = P_0 \mathcal{D}_1(0) \subseteq P_0 \langle 1_b, P_1 1_b, \ldots, P_{1:n-1} 1_b \rangle = \operatorname{im} X.$$

As $\mathcal{D}(0) = \mathbb{F}_2^n$, we have $\mathbb{F}_2^n \subseteq \operatorname{im} X$, and X is therefore invertible.

Conversely, we now assume that X is invertible, and prove by induction that the set of outputs after stage k that depend on the i^{th} input is

$$\mathcal{D}_k(i) = P_{k:n} i_b + \langle 1_b, P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle. \tag{77}$$

We already have $\mathcal{D}_{n+1}(i) = \{i_b\}$. Assuming (77) for $k+1$, and considering an element

$$j_b \in P_k \mathcal{D}_{k+1}(i) = P_{k:n} i_b + \langle P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle,$$

we have $j_b + P_{k:n} i_b \in \langle P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle$. As X is invertible, so is the matrix

$$P_{0:k-1}^{-1} X = \begin{pmatrix} P_{k:n-1} 1_b & \cdots & P_k 1_b & 1_b & P_{k-1}^{-1} 1_b & \cdots & P_{1:k-1}^{-1} 1_b \end{pmatrix}.$$

Its columns are linearly independent, and particularly,

$$1_b \notin \langle P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle.$$

Therefore, $j_b + P_{k:n} i_b + 1_b \notin \langle P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle$, thus $j_b + 1_b \notin P_k \mathcal{D}_{k+1}(i)$. This means that if a signal j_b that arrives on a DFT_2 depends on an input i ($j_b \in P_k \mathcal{D}_{k+1}(i)$), the other signal ($j_b + 1_b$) doesn't. As the output of a DFT_2 is the sum (resp. the difference) of these, and that an input i never appears on both terms of this operation, the dependency of both outputs on inputs is the union of the dependencies of the signals that arrive to this DFT_2. This yields (74), and a direct computation shows that

$$\begin{aligned}
\mathcal{D}_k(i) &= P_k \mathcal{D}_{k+1}(i) \cup (P_k \mathcal{D}_{k+1}(i) + 1_b) \\
&= (P_{k:n} i_b + \langle P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle) \cup \\
&\quad (P_{k:n} i_b + \langle P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle + 1_b) \\
&= P_{k:n} i_b + \langle 1_b, P_k 1_b, \ldots, P_{k:n-1} 1_b \rangle.
\end{aligned}$$

This yields (77), and $\mathcal{D}(0) = \mathbb{F}_2^n$ as a direct consequence. To be more precise, if (75) is satisfied at rank $k+1$, a signal j_b depending on the input i that arrives on the DFT_2 array of the k^{th} stage is in one of these cases:

- $j_b \in P_k \mathcal{D}^+_{k+1}(i)$, and j arrives on top of the DFT_2 (j is even, i.e. $j_b^T 1_b = 0$). In this case, both outputs of this DFT_2 depend "positively" on i: $\{j_b, j_b + 1_b\} \subseteq \mathcal{D}^+_k(i)$.
- $j_b \in P_k \mathcal{D}^-_{k+1}(i)$, and j arrives on top of the DFT_2 (j is even, i.e. $j_b^T 1_b = 0$). In this case, both outputs of this DFT_2 depend "negatively" on i: $\{j_b, j_b + 1_b\} \subseteq \mathcal{D}^-_k(i)$.
- $j_b \in P_k \mathcal{D}^+_{k+1}(i)$, and j arrives on the bottom of the DFT_2 (j is odd, i.e. $j_b^T 1_b = 1$). In this case, the top output depends "positively" on i: $j_b + 1_b \in \mathcal{D}^+_k(i)$, and the bottom output "negatively": $j_b \in \mathcal{D}^-_k(i)$.
- $j_b \in P_k \mathcal{D}^-_{k+1}(i)$, and j arrives on the bottom of the DFT_2 (j is odd, i.e. $j_b^T 1_b = 1$). In this case, the top output depends "negatively" on i: $j_b + 1_b \in \mathcal{D}^-_k(i)$, and the bottom output "positively": $j_b \in \mathcal{D}^+_k(i)$.

This yields (76) and (75) at rank k. As we have as well $\mathcal{D}^+_{n+1}(i) \cup \mathcal{D}^-_{n+1}(i) = \{i_b\} \cup \emptyset = \mathcal{D}_{n+1}(i)$, (75) holds for all k. □

7.4.2 *About condition (73)*

As mentioned earlier, we will derive a general expression for $W(P)$, assuming that it has its first row and columns filled with 1s. In the following lemma, we provide an equivalent condition for this assumption.

Lemma 14. *The following propositions are equivalent:*

- *The first row and the first column of $W(P)$ contain only 1s:*

$$\mathcal{D}^+(0) = \mathbb{F}_2^n, \text{ and} \tag{78}$$
$$0_b \in \mathcal{D}^+(i), \text{ for } 0 \leqslant i < 2^n. \tag{79}$$

- *The spreading matrix satisfies (73).*
- *No sequential product $P_{k:\ell}$ of the central matrices has a 1 in the bottom right corner, and the same holds for the inverse of $P_{k:\ell}$:*

$$1_b^T P_{k:\ell} 1_b = 0, \text{ for } 0 < k \leqslant \ell < n, \text{ and} \tag{80}$$
$$1_b^T P_{k:\ell}^{-1} 1_b = 0, \text{ for } 0 < k \leqslant \ell < n. \tag{81}$$

Proof. We start by showing the equivalence between the second and the third proposition. We consider the canonical basis $(e_1, \dots, e_n)$ of $\mathbb{F}_2^n$. We have, for all $0 < k, k' \leqslant n$,

$$\begin{aligned}
&e_k^T \begin{pmatrix} P_{0:n-1}^{-T} 1_b & P_{0:n-2}^{-T} 1_b & \dots & P_{0:1}^{-T} 1_b & P_0^{-T} 1_b \end{pmatrix}^T X e_{k'} \\
=&(P_{0:n-k}^{-T} 1_b)^T P_{0:n-k'} 1_b \\
=&1_b^T P_{1:n-k}^{-1} P_{1:n-k'} 1_b \\
=&\begin{cases} 1 & \text{if } k = k', \text{ or} \\ 1_b^T P_{n-k-1:n-k'} 1_b & \text{if } k > k', \text{ or} \\ 1_b^T P_{n-k'-1:n-k}^{-1} 1_b & \text{if } k' > k. \end{cases}
\end{aligned}$$

The nullity of the two last cases is equivalent to (73) on one side, and (80) and (81) on the other side.

We now consider the first proposition. We assume first that (78) holds, and will show that it implies (80). As $\mathbb{F}_2^n = \mathcal{D}^+(0) \subseteq \mathcal{D}(0)$, we can use the results of Lemma 13. Using (74), (75) and (76), we have

$$\begin{aligned}
\mathcal{D}_k^-(i) =& \mathcal{D}_k(i) \setminus \mathcal{D}_k^+(i) \\
=& (P_k\mathcal{D}_{k+1}(i) \cup (P_k\mathcal{D}_{k+1}(i) + 1_b)) \setminus \\
& \left(\begin{array}{l} (P_k\mathcal{D}_{k+1}^+(i) + 1_b) \cup \{j_b \in P_k\mathcal{D}_{k+1}^+(i) \mid j_b^T 1_b = 0\} \cup \\ \{j_b \in P_k\mathcal{D}_{k+1}^-(i) \mid j_b^T 1_b = 1\} \end{array} \right) \\
\supseteq& (P_k\mathcal{D}_{k+1}(i) + 1_b)) \setminus \left(P_k\mathcal{D}_{k+1}^+(i) + 1_b\right) \\
=& P_k\mathcal{D}_{k+1}^-(i) + 1_b
\end{aligned}$$

Therefore, the number of outputs depending "negatively" on a given input i can only increase within a stage:

$$|\mathcal{D}_k^-(i)| \geqslant |\mathcal{D}_{k+1}^-(i)|.$$

Particularly, for the first input, this means that $|\mathcal{D}^-(0)| = |\mathcal{D}_1^-(0)| \geqslant \cdots \geqslant |\mathcal{D}_n^-(0)|$. As $\mathcal{D}^-(0) = \emptyset$, we have $\mathcal{D}_k^+(0) = \mathcal{D}_k(0)$ for all k. Using again (74), (75) and (76), we have, for all k,

$$\begin{aligned}
\emptyset =& \mathcal{D}_k^-(0) \\
=& (P_k\mathcal{D}_{k+1}(0) \cup (P_k\mathcal{D}_{k+1}(0) + 1_b)) \setminus \\
& \left((P_k\mathcal{D}_{k+1}^+(0) + 1_b) \cup \{j_b \in P_k\mathcal{D}_{k+1}^+(0) \mid j_b^T 1_b = 0\}\right) \\
=& \{j_b \in P_k\mathcal{D}_{k+1}^+(0) \mid j_b^T 1_b = 1\},
\end{aligned}$$

Therefore, $j_b^T 1_b = 0$ for all k and all $j_b \in P_k\mathcal{D}_{k+1}^+(0) = P_k\mathcal{D}_{k+1}(0) = \langle P_k 1_b, P_k P_{k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle$, which yields (80).

We now consider that (79) is satisfied. This means that (78) holds for $W(P)^T = W((P_n^{-1}, \ldots, P_0^{-1}))$, and the same computation yields (81).

Finally, we suppose that (80) and (81) hold (and therefore (73), which allows to use Lemma 13). For $i = 0$, (76) becomes

$$\begin{aligned}
\mathcal{D}_k^+(0) =& (P_k\mathcal{D}_{k+1}^+(0) + 1_b) \cup \{j_b \in P_k\mathcal{D}_{k+1}^+(0) \mid j_b^T 1_b = 0\} \cup \\
& \{j_b \in P_k\mathcal{D}_{k+1}^-(0) \mid j_b^T 1_b = 1\}.
\end{aligned}$$

However, (80) yields that $j_b^T 1_b = 0$ in all cases. Therefore, we have

$$\mathcal{D}_k^+(0) = (P_k\mathcal{D}_{k+1}^+(0) + 1_b) \cup \{j_b \in P_k\mathcal{D}_{k+1}^+(0)\},$$

and a direct induction yields (78). The same argument with (81) and $W(P)^T$ yields (79). □

The following result is useless in the current form of the proof, but it is funny, and nobody will read this part anyway...

Lemma 15. *An n-ary quadratic form over $\mathbb{F}_2$ is linear if and only if the associated matrix is symmetric:*

$$i_b \mapsto i_b^T S i_b \text{ linear } \Leftrightarrow S \text{ symmetric.}$$

Proof. We consider the canonical basis $(e_0, \dots, e_{n-1})$ of $\mathbb{F}_2^n$, and a vector $x = \sum_i \lambda_i e_i$ of this space. If $S = (S_{i,j})$ is symmetric, then we have

$$\begin{aligned} x^T S x &= \sum_{i,j} \lambda_i \lambda_j S_{i,j} \\ &= \sum_i \lambda_i^2 S_{i,i} + \sum_{i>j} \lambda_i \lambda_j S_{i,j} + \sum_{i<j} \lambda_i \lambda_j S_{i,j} \\ &= \sum_i \lambda_i S_{i,i} + \sum_{i>j} \lambda_i \lambda_j S_{i,j} + \sum_{i<j} \lambda_i \lambda_j S_{j,i} \\ &= \sum_i \lambda_i S_{i,i} \\ &= \mathrm{Diag}(S)^T x. \end{aligned}$$

Conversely, if there exists $v \in \mathbb{F}_2^n$ such that for all $x \in \mathbb{F}_2^n$, $x^T S x = v^T x$, then we have, for all $0 \leqslant i, j < n$

$$\begin{aligned} 0 &= (e_i + e_j)^T S (e_i + e_j) + v^T (e_i + e_j) \\ &= e_i^T S e_j + e_j^T S e_i + e_i^T S e_i + e_j^T S e_j + v^T e_i + v^T e_j \\ &= S_{i,j} + S_{j,i} + v^T e_i + v^T e_j + v^T e_i + v^T e_j \\ &= S_{i,j} + S_{j,i}. \end{aligned}$$

□

7.4.3 *General expression of* $W(P)$

When X is invertible, we have $\mathcal{D}(i) = \mathcal{D}^+(i) \cup \mathcal{D}^-(i)$, which means that $W(P)$ is entirely determined by $\mathcal{D}^+$, for which we derive an expression in the following lemma.

Lemma 16. *If the spreading matrix satisfies* (73)*, then*

$$\mathcal{D}^+(i) = \ker\left(i_b^T P_{0:n}^T (XX^T)^{-1}\right) = \left\{j_b \mid i_b^T P_{0:n}^T (XX^T)^{-1} j_b = 0\right\}. \tag{82}$$

Proof. We assume that X satisfies (73). As X is invertible, the results of Lemma 13 and 14 can be used. Additionally, for $1 \leqslant k \leqslant n+1$, the vectors $\{1_b, P_k 1_b, P_{k:k+1} 1_b, \dots, P_{k:n-1} 1_b\}$ are linearly independent (as columns of the invertible matrix $P_{0:k-1}^{-1} X$), and we define on the $(n-k+1)$-dimensional space spawn by these vectors the linear mapping f_k as follows:

$$\begin{aligned} 1_b &\mapsto P_{k:n}^T 1_b \\ P_k 1_b &\mapsto P_{k+1:n}^T 1_b \\ &\vdots \\ P_{k:n-1} 1_b &\mapsto P_n^T 1_b. \end{aligned}$$

This mapping satisfies, for $1 \leqslant k \leqslant n$, and for all x in the domain of f_{k+1},

$$f_k(P_k x) = f_{k+1}(x). \tag{83}$$

Additionally, for $k = 1$, this mapping is defined over $\mathbb{F}_2^n$, and satisfies, for all $0 \leqslant x < 2^n$,

$$\begin{aligned} f_1(x_b) &= \begin{pmatrix} P_n^T 1_b & P_{n-1:n}^T 1_b & \cdots & P_{1:n}^T 1_b \end{pmatrix} \cdot \\ &\quad \begin{pmatrix} P_{1:n-1} 1_b & P_{1:n-2} 1_b \cdots & 1_b \end{pmatrix}^{-1} x_b \\ &= P_{0:n}^T \begin{pmatrix} P_{0:n-1}^{-T} 1_b & P_{0:n-2}^{-T} 1_b & \cdots & P_0^{-T} 1_b \end{pmatrix} \cdot \\ &\quad \begin{pmatrix} P_{0:n-1} 1_b & P_{0:n-2} 1_b & \cdots & P_0 1_b \end{pmatrix}^{-1} P_0 x_b \\ &= P_{0:n}^T (XX^T)^{-1} P_0 x_b. \end{aligned}$$

We define as well the vector s_k:

$$s_k = 1_b + P_k 1_b + P_{k:k+1} 1_b + \cdots + P_{k:n-1} 1_b.$$

We will now prove by induction that, for $1 \leqslant k \leqslant n+1$,

$$\mathcal{D}_k^+(i) = \{j_b \in \mathcal{D}_k(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + s_k) = 0\}. \tag{84}$$

We have, for $k = n+1$,

$$\begin{aligned} \mathcal{D}_{n+1}^+(i) &= \{i_b\} \\ &= \{j_b \in \mathcal{D}_{n+1}(i) \mid i_b^T f_{n+1}(j_b + I_n i_b + 0_b) = 0\}. \end{aligned}$$

If we assume that (84) is satisfied at rank $k+1$, i.e $\mathcal{D}_{k+1}^+(i) = \{j_b \in \mathcal{D}_{k+1}(i) \mid i_b^T f_{k+1}(j_b + P_{k+1:n} i_b + s_{k+1}) = 0\}$, we have, using (83),

$$\begin{aligned} P_k \mathcal{D}_{k+1}^+(i) &= \left\{ \begin{array}{l} j_b \in P_k \mathcal{D}_{k+1}(i) \mid \\ i_b^T f_{k+1}(P_k^{-1}(j_b + P_{k:n} i_b + P_k s_{k+1})) = 0 \end{array} \right\} \\ &= \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + P_k s_{k+1}) = 0\}. \end{aligned}$$

Using (75), we directly get

$$P_k \mathcal{D}_{k+1}^-(i) = \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + P_k s_{k+1}) = 1\}.$$

We can now compute $\mathcal{D}_k^+$ using (76):

$$\begin{aligned} \mathcal{D}_k^+(i) &= (P_k \mathcal{D}_{k+1}^+(i) + 1_b) \cup \{j_b \in P_k \mathcal{D}_{k+1}^+(i) \mid j_b^T 1_b = 0\} \cup \\ &\quad \{j_b \in P_k \mathcal{D}_{k+1}^-(i) \mid j_b^T 1_b = 1\} \\ &= (P_k \mathcal{D}_{k+1}^+(i) + 1_b) \cup \\ &\quad \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + P_k s_{k+1}) + j_b^T 1_b = 0\}. \end{aligned}$$

The term $j_b^T 1_b$ that appears for

$$j_b \in P_k \mathcal{D}_{k+1}(i) = P_{k:n} i_b + \langle P_k 1_b, P_{k:k+1} 1_b, \ldots, P_{k:n-1} 1_b \rangle$$

satisfies, using (80), $j_b^T 1_b = i_b^T P_{k:n}^T 1_b = i_b^T f_k(1_b)$. Therefore:

$$\begin{aligned} \mathcal{D}_k^+(i) &= (P_k \mathcal{D}_{k+1}^+(i) + 1_b) \cup \\ &\quad \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + P_k s_{k+1}) + i_b^T f_k(1_b) = 0\} \\ &= (P_k \mathcal{D}_{k+1}^+(i) + 1_b) \cup \\ &\quad \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + s_k) = 0\} \\ &= \{j_b \in P_k \mathcal{D}_{k+1}(i) + 1_b \mid i_b^T f_k(j_b + P_{k:n} i_b + s_k) = 0\} \cup \\ &\quad \{j_b \in P_k \mathcal{D}_{k+1}(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + s_k) = 0\} \\ &= \{j_b \in \mathcal{D}_k(i) \mid i_b^T f_k(j_b + P_{k:n} i_b + s_k) = 0\}, \end{aligned}$$

which yields the result.

We can now provide a first expression for $\mathcal{D}^+$, using (83):

$$\begin{aligned}\mathcal{D}^+(i) &= P_0\mathcal{D}_1^+(i)\\ &= P_0\{j_b \mid i_b^T f_1(j_b + P_{1:n}i_b + s_1) = 0\}\\ &= P_0\{j_b \mid i_b^T P_{0:n}^T (XX^T)^{-1}(P_0 j_b + P_{0:n}i_b + P_0 s_1) = 0\}\\ &= \{j_b \mid i_b^T P_{0:n}^T (XX^T)^{-1}(j_b + P_{0:n}i_b + P_0 s_1) = 0\}.\end{aligned}$$

The last step consists in showing that the mapping

$$g : i_b \mapsto i_b^T P_{0:n}^T (XX^T)^{-1}(P_{0:n}i_b + P_0 s_1)$$

is null. We consider the canonical basis $(e_1, \dots, e_n)$ of $\mathbb{F}_2^n$, and a vector $x = \sum_i \lambda_i P_{0:n}^{-1} X e_i$ of this space. A direct computation yields:

$$\begin{aligned}g(x) =& x^T P_{0:n}^T (XX^T)^{-1}(P_{0:n}x + P_0 s_1)\\ =& (X^{-1}P_{0:n}x)^T X^{-1}(P_{0:n}x + P_0 s_1)\\ =& \left(X^{-1}P_{0:n}\sum_i \lambda_i P_{0:n}^{-1} X e_i\right)^T X^{-1} \cdot \left(P_{0:n}\sum_j \lambda_j P_{0:n}^{-1} X e_j + \sum_k X e_k\right)\\ =& \sum_i \lambda_i e_i^T \sum_j (\lambda_j + 1) e_j\\ =& \sum_{i,j} \lambda_i(\lambda_j + 1) e_i^T e_j\\ =& \sum_{i=j} \lambda_i(\lambda_j + 1) e_i^T e_j + \sum_{i\neq j} \lambda_i(\lambda_j + 1) e_i^T e_j\\ =& 0.\end{aligned}$$

□

7.4.4 *Proof of Lemma 12*

The proof of Lemma 12 is now straightforward.

If $P \in \mathcal{P}$, then $W(P)$ has its first column and row filled up with 1s. Lemma 14 ensures (73), and we can use Lemma 16. Identifying (82) with the definition (25) of H_{2^n} yields that $P_{0:n}^T = XX^T$, i.e. that (72) is satisfied.

Conversely, if (72) and (73) are satisfied, $W(P) = H_{2^n}$ is a direct consequence of Lemma 16.

7.5 PROOF OF THEOREM 3

We first assume that $(P_0, \dots, P_n) \in \mathcal{P}$, and construct a set of matrices satisfying (31). We define $B = X^{-1}$, and, by induction, the matrices

$$\tilde{Q}_i = \begin{cases} B^{-1} \cdot P_0, & \text{for } i = 1,\\ C_n^{-1} \cdot \tilde{Q}_{i-1} \cdot P_{i-1}, & \text{for } 1 < i \leqslant n.\end{cases}$$

By construction, these matrices satisfy, for $0 < i \leqslant n$,

$$\tilde{Q}_i = C_n^{1-i} X^{-1} P_{0:i-1}.$$

Using (72), we get:

$$P_i = \begin{cases} B \cdot \tilde{Q}_1, & \text{for } i = 0, \\ \tilde{Q}_i^{-1} \cdot C_n \cdot \tilde{Q}_{i+1}, & \text{for } 0 < i < n, \\ \tilde{Q}_n^{-1} \cdot C_n \cdot \tilde{B}^T, & \text{for } i = n. \end{cases}$$

The last step consists in showing that, for $0 < i \leqslant n$, there exists a matrix $Q_i \in GL_{n-1}(\mathbb{F}_2)$ such that $\tilde{Q}_i = \begin{pmatrix} Q_i & \\ & 1 \end{pmatrix}$, or equivalently, that $\tilde{Q}_i 1_b = \tilde{Q}_i^T 1_b = 1_b$:

$$\begin{aligned} \tilde{Q}_i 1_b &= C_n^{1-i} X^{-1} P_{0:i-1} 1_b \\ &= C_n^{1-i} \begin{pmatrix} P_{0:n-1} 1_b & \dots & P_{0:1} 1_b & P_0 1_b \end{pmatrix}^{-1} P_{0:i-1} 1_b \\ &= 1_b. \end{aligned}$$

A similar computation, using (73), shows that $\tilde{Q}_i^T 1_b = 1_b$.

Conversely, if a set of matrices satisfy (31), a direct computation (with [33]) shows that (72) and (73) are satisfied. Thus. $P \in \mathcal{P}$.

8

CONCLUSION AND FUTURE WORK

"Optimal" is often misused as a synonym for "good", or "best so far". But claiming optimality goes beyond the superlative; one has to show that no better solution can ever be found. In this dissertation, we demonstrated the optimality of our solutions in two different ways:

- For streaming permutations, we defined in Chapter 2 a novel metric, the *routing entropy*. Using a proof *à la Shannon*, we showed that this measure is in fact a lower bound on the number of multiplexers that an implementation of this streaming permutation contains (Theorem 1). Then, in the particular case of *linear permutations*, we proposed a systematic method to design an implementation matching this bound, thus characterizing exactly their routing complexity. Similarly, we described a second architecture capable of minimizing both the latency and the number of RAM banks used, while using the minimal number of multiplexers with respect to this constraint, and described precisely the permutations for which all these measures could simultaneously be minimized. The method we used involved a novel matrix factorization algorithm (Chapter 6), similar to a block LU decomposition with a minimization of the ranks of sub-diagonal matrices.
- For streaming Walsh-Hadamard transforms, we fully characterized all possible linearly permuted algorithms in Chapter 7 (Theorem 3), and employed in Chapter 4 advanced search techniques to find in this space those that have the best implementation cost.

We employed these theoritical advances in very concrete applications. Namely:

- We described in Chapter 3 a method to design a datapath capable of realizing different fixed streamed linear permutations. Taking a folded Pease fast Fourier transform as an example, we showed that "fusing" the internal shuffle and the bit reversal yields a significant reduction in terms of RAM consumption. This new architecture offers novel Pareto-optimal tradeoffs between performance and logic/memory across an entire design space of FFTs.
- We proposed in Chapter 5 a generator for streaming FFTs, WHTs and sorting networks. This generator follows a principled design using the state-of-the-art language features in Scala, and employs three embedded DSLs and the concept of staging to enable flexibility and automatic optimizations on different levels.

In addition to these results, this research produced an artifact available at [67] and [66].

Future work. The research presented in this dissertation satisfies the goal we pursued, providing optimality results in streaming permutations and transforms. We give a list of potential future directions:

- Most signal processing algorithms use linear permutations exclusively. However, many permutations are not linear, and in this case, the bound provided in Theorem 1 may not be sharp. A natural question arising would be to know whether

an implementation matching this bound exists, even for general streaming permutations, or if a higher bound can be found.

- For the Walsh-Hadamard transform, our results for small sizes strongly indicate that there is a class of yet undiscovered, and non-obvious algorithms with reduced RAM and logic requirements in streaming implementation. However, the space for bigger WHTs is too large to be exhaustively searched. A challenge would consist in extrapolating the optimal solutions found for small sizes to all sizes.
- The idea behind the characterization of the space of WHT algorithms is in principle applicable to other regular algorithms. Particularly, the discrete Fourier transform would be a good candidate for such an exploration.
- The principles we used to develop our generator can be used for other applications. An obvious line for future work is to expand our generator to other domains.

BIBLIOGRAPHY

[1] Berkin Akin, Franz Franchetti, and James C. Hoe. "FFTs with Near-Optimal Memory Access Through Block Data Layouts: Algorithm, Architecture and Design Automation." In: *Journal of Signal Processing Systems* 85.1 (2015), pp. 67–82 (cit. on p. 2).

[2] J. Astola and D. Akopian. "Architecture-Oriented Regular Algorithms for Discrete Sine and Cosine Transforms." In: *IEEE Transactions on Signal Processing* 47.4 (1999), pp. 1109–1124 (cit. on p. 2).

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a Scala embedded language." In: *Proc. Design Automation Conference (DAC)*. 2012, pp. 1216–1225 (cit. on p. 86).

[4] Ken Edward Batcher. "Sorting Networks and Their Applications." In: *Proc. Spring Joint Computer Conference*. Vol. 32. AFIPS. 1968, pp. 307–314 (cit. on pp. 2, 70, 87).

[5] K. G. Beauchamp. *Applications of Walsh and related functions with an introduction to sequency theory*. Academic Press, 1985 (cit. on p. 53).

[6] Váaclav Edvard Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965 (cit. on pp. 2, 34).

[7] Joël Busset, Florian Perrodin, Peter Wellig, Beat Ott, Kurt Heutschi, Torben Rühl, and Thomas Nussbaumer. "Detection and tracking of drones using advanced acoustic cameras." In: *Proc. SPIE* 9647 (2015) (cit. on p. vii).

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems." In: *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2011, pp. 33–36 (cit. on p. 86).

[9] D. Carlson, E. Haynsworth, and T. Markham. "A Generalization of the Schur Complement by Means of the Moore-Penrose Inverse." In: *SIAM Journal on Applied Mathematics* 26.1 (1974), pp. 169–175 (cit. on p. 94).

[10] Ren Chen and V.K. Prasanna. "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations." In: *Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8 (cit. on pp. 33, 34).

[11] Ren Chen and Viktor K. Prasanna. "Optimal circuits for parallel bit reversal." In: *Proc. Design Automation Conference (DAC)*. 2017 (cit. on pp. 5, 33, 35).

[12] Ren Chen, Sruja Siriyal, and Viktor Prasanna. "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA." In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. Monterey, California, USA, 2015, pp. 240–249 (cit. on pp. 5, 33, 34, 87).

[13] E. Chow and Y. Saad. "Approximate Inverse Techniques for Block-Partitioned Matrices." In: *SIAM Journal on Scientific Computing* 18.6 (1997), pp. 1657–1675 (cit. on p. 92).

[14] Danny Cohen. "Simplified control of FFT hardware." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24.6 (1976), pp. 577–579 (cit. on p. 2).

[15] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." In: *Mathematics of Computation* 19.90 (1965), pp. 297–301 (cit. on pp. 1, 55).

[16] Ainhoa Cortés, Igone Vélez, and Juan F. Sevillano. "Radix r^k FFTs: Matricial Representation and SDC/SDF Pipeline Implementation." In: *IEEE Transactions on Signal Processing* 57.7 (2009), pp. 2824–2839 (cit. on p. 2).

[17] Richard W. Cottle. "Manifestations of the Schur Complement." In: *Linear Algebra and its Applications* 8.3 (1974), pp. 189 –211 (cit. on p. 92).

[18] James W. Demmel, Nicholas J. Higham, and Robert S. Schreiber. "Block LU Factorization." In: *Numerical Linear Algebra with Applications* 2.2 (1992), pp. 173–190 (cit. on p. 92).

[19] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27 (cit. on p. 5).

[20] Robert W. Floyd. "Algorithm 97: Shortest Path." In: *Commun. ACM* 5.6 (1962), p. 345 (cit. on p. 59).

[21] Jean Baptiste Joseph Fourier. *Théorie analytique de la chaleur*. Chez Firmin Didot, père et fils, 1822 (cit. on p. 1).

[22] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. "Operator Language: A Program Generation Framework for Fast Kernels." In: *IFIP Working Conference on Domain Specific Languages (DSL WC)*. Vol. 5658. Lecture Notes in Computer Science. Springer, 2009, pp. 385–410 (cit. on p. 69).

[23] Mario Garrido. "Multiplexer and Memory-efficient circuits for parallel bit reversal." In: *IEEE Transactions on Circuits and Systems II (TCAS-II)* (2018) (cit. on pp. 4, 33, 35).

[24] Mario Garrido, Jesús Grajal, and Oscar Gustafsson. "Optimum circuits for bit-dimension permutations." In: *IEEE Transactions on Very Large Scale Integration Systems* (2019) (cit. on p. 5).

[25] Mario Garrido, Miguel Ángel Sánchez, María Luisa López-Vallejo, and Jesùs Grajal. "A 4096-Point Radix-4 Memory-Based FFT Using DSP Slices." In: *IEEE Transactions on Very Large Scale Integration Systems* 25.1 (2017), pp. 375–379 (cit. on pp. 2, 49).

[26] Johann Carl Friedrich Gauß. "Theoria interpolationis methodo nova tractata." In: *Werke* Band 3 (1866), pp. 265–327 (cit. on p. 1).

[27] Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. "Making domain-specific hardware synthesis tools cost-efficient." In: *Proc. International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 120–127 (cit. on p. 86).

[28] Nithin George, HyoukJoong Lee, David Novo, Muhsen Owaida, David Andrews, Kunle Olukotun, and Paolo Ienne. "Automatic support for multi-module parallelism from computational patterns." In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8 (cit. on p. 86).

[29] Jacques Hadamard. "Résolution d'une question relative aux déterminants." In: *Bulletin des sciences mathématiques* 17 (1893), pp. 240–246 (cit. on pp. 2, 53).

[30] Shousheng He and Mats Torkelson. "A new approach to pipeline FFT processor." In: *Proc. Parallel Processing Symposium (IPPS)*. 1996, pp. 766–770 (cit. on p. 2).

[31] Tuomas Järvinen, Perttu Salmela, Harri Sorokin, and Jarmo Takala. "Stride permutation networks for array processors." In: *Proc. International Conference on Application-Specific Systems, Architectures and Processors Proceedings (ASAP)*. 2004, pp. 376–386 (cit. on pp. 4, 35, 49).

[32] Byung G. Jo and Myung H. Sunwoo. "New continuous-flow mixed-radix (CFMR) FFT Processor using novel in-place strategy." In: *IEEE Transactions on Circuits and Systems I* 52.5 (2005), pp. 911–919 (cit. on pp. 2, 49).

[33] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures." In: *Circuits, Systems and Signal Processing* 9.4 (1990), pp. 449–500 (cit. on pp. 55, 68, 124).

[34] Jeremy Johnson and Michael Andrews. "Statistical Evaluation of a Self-Tuning Vectorized Library for the Walsh-Hadamard Transform." In: *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*. 2008 (cit. on p. 53).

[35] Jeremy Johnson and Markus Püschel. "In Search of the Optimal Walsh-Hadamard Transform." In: *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Vol. 6. 2000, pp. 3347–3350 (cit. on pp. 53, 57, 59).

[36] Alan H. Karp. "Bit Reversal on Uniprocessors." In: *SIAM Review* 38.1 (1996), pp. 1–26 (cit. on p. 21).

[37] Donald Ervin Knuth. *The Art of Computer Programming, 2Nd Ed. (Addison-Wesley Series in Computer Science and Information*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978 (cit. on pp. 11, 12, 87).

[38] Thaddeus Koehn and Peter Athanas. "Arbitrary streaming permutations with minimum memory and latency." In: *Proc. International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–6 (cit. on pp. 4, 5, 15, 33, 38, 46).

[39] Pinit Kumhom, Jeremy R. Johnson, and Prawat Nagvajara. "Design, optimization, and implementation of a universal FFT processor." In: *Proc. International ASIC/SOC Conference (ASIC)*. 2000, pp. 182–186 (cit. on p. 2).

[40] Jacques Lenfant and Serge Tahé. "Permuting data with the Omega network." In: *Acta Informatica* 21.6 (1985), pp. 629–641 (cit. on pp. 2, 22–24, 34).

[41] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977 (cit. on p. 53).

[42] Geoffrey Mainland and Jeremy Johnson. "A Haskell Compiler for Signal Transforms." In: *Proc. International Conference on Generative Programming: Concepts and Experiences (GPCE)*. 2017, pp. 219–232 (cit. on p. 86).

[43] Stéphane Mallat. *Traitement du signal*. École polytechnique, Département de Mathématiques appliquées, 2000 (cit. on p. 1).

[44] Peter A. Milder. *Spiral DFT/FFT IP Core Generator*. `http://www.spiral.net/hardware/dftgen.html`. 2008 (cit. on pp. 45, 49).

[45] Peter A. Milder, James C. Hoe, and Markus Püschel. "Automatic Generation of Streaming Datapaths for Arbitrary Fixed Permutations." In: *Proc. Design, Automation and Test in Europe (DATE)*. 2009, pp. 1118–1123 (cit. on pp. 4, 15, 31–33, 35, 38, 41, 46, 49).

[46] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. "Linear Transforms: From Math to Efficient Hardware." In: *Workshop on High-Level Synthesis colocated with DAC*. 2008 (cit. on pp. 2, 5).

[47] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. "Computer Generation of Hardware for Linear Digital Signal Processing Transforms." In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17.2 (2012), 15:1–15:33 (cit. on pp. 2–5, 37, 45, 46, 49, 68, 72, 82, 85, 86).

[48] Rene Mueller, Jens Teubner, and Gustavo Alonso. "Sorting Networks on FPGAs." In: *The VLDB Journal* 21.1 (2012), pp. 1–23 (cit. on p. 87).

[49] David Nassimi and Sartaj Sahni. "A Self-Routing Benes Network and Parallel Permutation Algorithms." In: *IEEE Transactions on Computers* 30.5 (1981), pp. 332–340 (cit. on pp. 2, 23, 34).

[50] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel. "Automatic Generation of Customized Discrete Fourier Transform IPs." In: *Proc. Design Automation Conference (DAC)*. 2005, pp. 471–474 (cit. on pp. 2, 4, 55).

[51] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008 (cit. on p. 7).

[52] Georg Ofenbeck. "Generic programming in space and time." PhD thesis. ETH Zurich, 2017 (cit. on p. 86).

[53] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. "Staging for Generic Programming in Space and Time." In: *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 2017, pp. 15–28 (cit. on p. 77).

[54] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries." In: *Proc. International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 2013, pp. 125–134 (cit. on pp. 68, 77).

[55] Stéphane Olivier. *Physique 2e année PC PC**. Lavoisier / Tec & Doc, 2004 (cit. on p. 1).

[56] J. Ortiz and D. Andrews. "A configurable high-throughput linear sorter system." In: *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 2010, pp. 1–8 (cit. on p. 87).

[57] Kevin J. Page, Jeanette F. Arrigo, and Paul M. Chau. "Reconfigurable-hardware-based digital signal processing for wireless communications." In: *Advanced Signal Processing Algorithms, Architectures and Implementations*. Ed. by Proc. SPIE. Vol. 3162. 7. 1997, pp. 529–540 (cit. on pp. 2, 18, 21).

[58] Kevin J. Page and Paul M. Chau. "Folding large regular computational graphs onto smaller processor arrays." In: *Advanced Signal Processing Algorithms, Architectures and Implementations*. Ed. by Proc. SPIE. Vol. 2846. 7. 1996, pp. 383–394 (cit. on p. 4).

[59] Keshab K. Parhi. "Systematic synthesis of DSP data format converters using lifetime analysis and forward-backward register allocation." In: *IEEE Transactions on Circuits and Systems II (TCAS-II)* 39.7 (1992), pp. 423–440 (cit. on pp. 4, 35, 49).

[60] Marshall C. Pease. "An Adaptation of the Fast Fourier Transform for Parallel Processing." In: *Journal of the ACM* 15.2 (1968), pp. 252–264. ISSN: 0004-5411 (cit. on pp. 2, 11, 53, 55, 69).

[61] Marshall C. Pease. "The Indirect Binary n-Cube Microprocessor Array." In: *IEEE Transactions on Computers* 26.5 (1977), pp. 458–473 (cit. on pp. 2, 22, 34).

[62] O. Port and Y. Etsion. "DFiant: A dataflow hardware description language." In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–4 (cit. on p. 86).

[63] Markus Püschel, Peter A. Milder, and James C. Hoe. "Permuting Streaming Data Using RAMs." In: *Journal of the ACM* 56.2 (2009), 10:1–10:34 (cit. on pp. 4, 5, 21, 22, 24, 29, 31–34, 41, 45, 46, 95).

[64] Markus Püschel et al. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93.2 (2005), pp. 232–275 (cit. on pp. 2, 68, 86).

[65] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." In: *Commun. ACM* 55.6 (June 2012), pp. 121–130 (cit. on pp. 7, 76, 86).

[66] François Serre. *DFT and Streamed Linear Permutation Generator for Hardware*. https://github.com/fserre/sgen. 2018 (cit. on pp. 49, 87, 125).

[67] François Serre. *SGen - A Streaming Hardware Generator*. https://acl.inf.ethz.ch/research/hardware/. 2018 (cit. on pp. 87, 125).

[68] François Serre, Thomas Holenstein, and Markus Püschel. "Optimal Circuits for Streamed Linear Permutations Using RAM." In: *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2016, pp. 215–223 (cit. on pp. 11, 68, 85).

[69] François Serre and Markus Püschel. "Generalizing block LU factorization: A Lower-Upper-Lower block triangular decomposition with minimal off-diagonal ranks." In: *Linear Algebra and its Applications* 509 (2016), pp. 114–142 (cit. on p. 91).

[70] François Serre and Markus Püschel. "Characterizing and Enumerating Walsh-Hadamard Transform Algorithms." In: *CoRR* abs/1710.08029 (2017). arXiv: 1710.08029 (cit. on p. 115).

[71] François Serre and Markus Püschel. "Optimal Streamed Linear Permutations." In: *Proc. Symposium on Computer Arithmetic (ARITH)*. 2017, pp. 60–61 (cit. on p. 11).

[72] François Serre and Markus Püschel. "A DSL-based FFT hardware generator in Scala." In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 315–322 (cit. on p. 67).

[73] François Serre and Markus Püschel. "Memory-Efficient Fast Fourier Transform on Streaming Data by Fusing Permutations." In: *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2018, pp. 219–228 (cit. on p. 37).

[74] François Serre and Markus Püschel. "DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks." In: *Transactions on Reconfigurable Technology and Systems* (2019). In press. (cit. on p. 67).

[75] François Serre and Markus Püschel. "In search of the optimal Walsh-Hadamard transform for streamed parallel processing." In: *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2019 (cit. on p. 53).

[76] Claude Elwood Shannon. "A mathematical theory of communication." In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423 (cit. on p. 17).

[77] G. Steidl and M. Tasche. "A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms." In: *Mathematics of Computation* 56.193 (1991), pp. 281–296 (cit. on p. 14).

[78] David Steinberg. "Invariant Properties of the Shuffle-Exchange and a Simplified Cost-Effective Version of the Omega Network." In: *IEEE Transactions on Computers* 32.5 (1983), pp. 444–450 (cit. on pp. 2, 34).

[79] Harold S. Stone. "Parallel Processing with the Perfect Shuffle." In: *IEEE Transactions on Computers* C-20.2 (1971), pp. 153–161 (cit. on pp. 2, 70, 71).

[80] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. "OptiML: an implicitly parallel domain-specific language for machine learning." In: *Proc. International Conference on Machine Learning (ICML)*. 2011, pp. 609–616 (cit. on p. 86).

[81] J. H. Takala, T. S. Jarvinen, and H. T. Sorokin. "Conflict-free parallel memory access scheme for FFT processors." In: *Proc. International Symposium on Circuits and Systems (ISCAS)*. Vol. 4. 2003, pp. 524–527 (cit. on p. 2).

[82] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. Siam, 1992 (cit. on p. 55).

[83] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc." In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 60–76 (cit. on p. 77).

[84] Abraham Waksman. "A permutation network." In: *Journal of the ACM* 15.1 (1968), pp. 159–163 (cit. on pp. 2, 18, 34, 45).

[85] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. "SPL: A Language and Compiler for DSP Algorithms." In: *Programming Languages Design and Implementation (PLDI)*. 2001, pp. 298–308 (cit. on p. 68).

[86] Rao K. Yarlagadda and John E. Hershey. *Hadamard Matrix Analysis and Synthesis*. Springer Us, 2012 (cit. on p. 53).

[87] Fuzhen Zhang. *The Schur Complement and its Applications*. Vol. 4. Springer, 2006 (cit. on pp. 92, 94, 95).

[88] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. "Computer Generation of Streaming Sorting Networks." In: *Design Automation Conference (DAC)*. 2012, pp. 1245–1253 (cit. on pp. 2, 86).

[89] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. "Streaming Sorting Networks." In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 21.4 (2016) (cit. on pp. 2, 5, 11, 12, 69–71, 85, 86).

FRANÇOIS SERRE

Date of birth: 6th January 1986
Nationality: French

EDUCATION

2013 - 2019	**ETH Zurich** PhD in Computer Science, Prof. Dr. Püschel
2010 - 2012	**ETH Zurich** MSc in Computer Science
2007 - 2012	**Ecole Polytechnique, Palaiseau, France** Dipl. Ing., X2007
2004 - 2007	**Lycée Louis-Le-Grand, Paris, France** Classes préparatoires aux grandes écoles, PCSI/PC*

WORK EXPERIENCE

04/2010 - 09/2010	**Siemens Corporate Research, Princeton, NJ, US** Research internship
08/2009 - 09/2010	**Ferrum Lasercut, Berlin, Germany** Blue-collar internship
09/2007 - 04/2008	**Gendarmerie Nationale, Tours, France** Second lieutenant