Streaming Sorting Networks

MARCELA ZULUAGA, Department of Computer Science, ETH Zurich
PETER MILDER, Department of Electrical and Computer Engineering, Stony Brook University
MARKUS PÜSCHEL, Department of Computer Science, ETH Zurich

Sorting is a fundamental problem in computer science and has been studied extensively. Thus, a large variety of sorting methods exist for both software and hardware implementations. For the latter, there is a trade-off between the throughput achieved and the cost (i.e., the logic and storage invested to sort n elements). Two popular solutions are bitonic sorting networks with $O(n \log^2 n)$ logic and storage, which sort n elements per cycle, and linear sorters with $O(n)$ logic and storage, which sort n elements per n cycles. In this article, we present new hardware structures that we call *streaming sorting networks*, which we derive through a mathematical formalism that we introduce, and an accompanying domain-specific hardware generator that translates our formal mathematical description into synthesizable RTL Verilog. With the new networks, we achieve novel and improved cost-performance trade-offs. For example, assuming that n is a two-power and w is any divisor of n, one class of these networks can sort in n/w cycles with $O(w \log^2 n)$ logic and $O(n \log^2 n)$ storage; the other class that we present sorts in $n \log^2 n/w$ cycles with $O(w)$ logic and $O(n)$ storage. We carefully analyze the performance of these networks and their cost at three levels of abstraction: (1) asymptotically, (2) exactly in terms of the number of basic elements needed, and (3) in terms of the resources required by the actual circuit when mapped to a field-programmable gate array. The accompanying hardware generator allows us to explore the entire design space, identify the Pareto-optimal solutions, and show superior cost-performance trade-offs compared to prior work.

CCS Concepts:• **Theory of computation** → *Sorting and searching;*• **Hardware** → *Logic circuits; Reconfigurable logic and FPGAs; High-level and register-transfer level synthesis; Hardware description languages and compilation;*

Additional Key Words and Phrases: Hardware Sorting, Design Space Exploration, HDL Generation

ACM Reference Format:
Marcela Zuluaga, Peter Milder, and Markus Püschel, 2016. Streaming sorting networks. *ACM Trans. Des. Autom. Electron. Syst.* 21, 4, Article 55 (May 2016), 28 pages.
DOI: http://dx.doi.org/10.1145/2854150

1. INTRODUCTION

Sorting is one of the fundamental operations in computing and is a crucial component in many applications. Because of its importance, there has been extensive research on fast algorithms for sorting in software and hardware [Knuth 1968]. In software, numerous algorithms with different characteristics exist that achieve the optimal asymptotic runtime of $\Theta(n \log n)$. In hardware, the situation is different, as there is a trade-off between the cost (e.g., area) and the performance achieved. For example, the two most common sorting methods in hardware are the bitonic sorting network and the linear sorter. For a list of length n, the former requires $\Theta(n \log^2 n)$ logic and can sort n elements per cycle; the latter requires $\Theta(n)$ logic and sorts n elements in n cycles.

In this article, we present a class of flexible hardware structures that we call *streaming sorting networks* and an accompanying domain-specific hardware generation tool that can automatically create synthesizable register-transfer level (RTL) Verilog for any design in the class. We systematically derive streaming sorting networks using a mathematical formalism, carefully analyze (asymptotically and exactly) their cost and performance, and show that they can offer both novel trade-offs and im-

Author's addresses: M. Zuluaga and M. Püschel, Department of Computer Science, ETH Zurich, Universitätsstrasse 6, 8092 Zurich, Switzerland; P. Milder, Electrical and Computer Engineering Department, Stony Brook University, Stony Brook, NY 11794-2350, USA.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. 1084-4309/2016/05-ART55 $15.00
DOI: http://dx.doi.org/10.1145/2854150

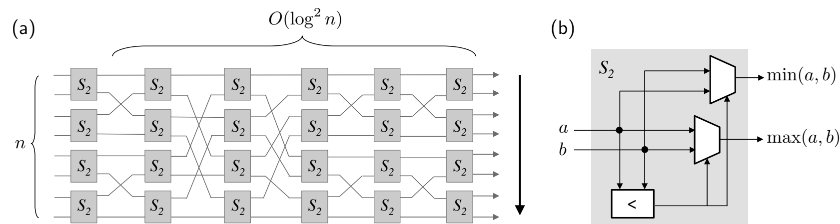


Fig. 1. (a) Bitonic sorting network for an input set size $n = 8$. (b) The two-input sorter S_2 is the building block in sorting networks.

provements (again in asymptotic and exact terms) over prior solutions. We then describe how our hardware synthesis tool compiles our mathematical representation into Verilog. Using this tool we experimentally confirm our analysis and show the benefits of our new networks.

This work significantly extends our prior work [Zuluaga et al. 2012a], where we introduced the concept of streaming sorting networks and an associated domain-specific hardware generation tool that produces their design spaces.

1.1. Background and Related Work

In this section we review the most important solutions for sorting in hardware. We begin by defining necessary key terms for describing properties of the various designs.

Definition 1.1 (*Streaming*). Consider a hardware design that must take in n data words as input (and produces n data words as output). If the design receives (and emits) these elements at a rate of w words per clock cycle over n/w consecutive cycles, this design is called streaming. We call w the streaming width of the design.

By implementing designs with streaming width w less than n (the number of words to be sorted), the costs and I/O bandwidth of the system are reduced.

Definition 1.2 (*Fully Streaming*). A design that can be constantly fed with an input stream with no waiting time between two consecutive input sets is called a fully streaming design.

A fully streaming sorting network with streaming width w will have a throughput of w words per cycle. If a design is not fully streaming, then its throughput will be less than w words per cycle. Typically, we would expect a fully streaming design to require more resources than a design that is not fully streaming, because it must process more data in the same amount of time.

Next, we will use these definitions to characterize prior work on sorting hardware. We classify this work into three categories: sorting networks, linear sorters, and other approaches. The key solutions within these categories are characterized in Table I, which shows the asymptotic logic and storage cost in each case. Additionally, it indicates the supported streaming width and whether the architecture is fully streaming. In all cases, the number of input values is a power of two: $n = 2^t$.

Sorting networks. Sorting networks process in parallel a list of n input elements through several reordering stages such that the input list is sorted at the final stage. Each stage performs parallel pairwise comparisons followed by a data permutation. The number of stages required to obtain a sorted list and the number of comparison operations per stage determine the cost of the network.

Figure 1 shows an example of an eight-element sorting network. The operator S_2 sorts two inputs into ascending order. Sorting networks can achieve very high throughput, as they are fully streaming designs with streaming width n. They are typically pipelined to maintain a reasonable clock speed and then require $O(n \log^2 n)$ flip-flops for storage.

Many sorting networks have been proposed in the literature [Knuth 1968; Leighton 1992]. Figure 1 shows a *bitonic* sorting network [Batcher 1968]. These can be constructed recursively and have a very regular structure. They sort n elements in $\log_2 n(\log_2 n + 1)/2$ stages of $n/2$ parallel

Table I. Different Sorting Network Architectures and Their Key Features.

Architecture	Streaming Width	Logic	Storage (for a Pipelined Architecture)	Storage Type	Fully Streaming
Bitonic and odd-even sorting network [Batcher 1968]	n	$O(n \log^2 n)$	$O(n \log^2 n)$	Flip-flop	Yes
Folded bitonic sorting network [Stone 1971]	n	$O(n)$	$O(n)$	Flip-flop	No
Odd-even transposition sorting network [Knuth 1968]	n	$O(n^2)$	$O(n^2)$	Flip-flop	Yes
Folded odd-even transposition sorting network [Knuth 1968]	n	$O(n)$	$O(n)$	Flip-flop	No
AKS sorting network [Ajtai et al. 1983]	n	$O(n \log n)$	$O(n \log n)$	Flip-flop	Yes
Linear sorter [Perez-Andrade et al. 2009; Lee and Tsai 1995]	1	$O(n)$	$O(n)$	Flip-flop	Yes
Interleaved linear sorter (ILS) [Ortiz and Andrews 2010]	$1 \leq w \leq n$	$O(wn)$	$O(wn)$	Flip-flop	Yes
Shear-sort (2D mesh) [Scherson and Sen 1989]	n	$O(n)$	$O(n)$	Flip-flop	No
Streaming sorting network [This article]	$2 \leq w < n$	$O(w \log^2 n)$	$O(n \log^2 n)$	RAM	Yes
Folded streaming sorting network [This article]	$2 \leq w < n$	$O(w)$	$O(n)$	RAM	No

S_2 operations, thus requiring $O(n \log^2 n)$ comparison operations. Batcher also proposed odd-even sorting networks that are less regular but require slightly fewer comparison operations; these have been used, for example, in Mueller et al. [2012] on FPGAs.

Stone [1971] derived bitonic sorting networks with constant geometry. This means that two consecutive comparison stages of $n/2$ parallel comparisons are always connected with the same permutation—the so-called perfect shuffle. Thus the network can be *folded* to only instantiate one comparison stage and shuffle, through which the data is cycled $O(\log^2 n)$ times. The direction of the comparison operations is switched during the phases, which requires a small area overhead for control logic.

Similar efforts to fold or regularize bitonic sorting networks have been proposed later. Bilardi and Preparata [1984] implement bitonic sorting using various configurations of the pleated cube connection cycle interconnection network. Dowd et al. [1989] propose the periodic balanced sorting network, which achieves a different granularity of reuse with a block of $\log n$ stages that can be reused $\log n$ times. On the other hand, Layer et al. [2007] proposed a bitonic sorting network composed of $\log_2 n$ stages that recirculate the data several times within each stage.

A constraint of most parallel sorting architectures with a fixed computational flow, such as sorting networks, is that they must always sort input sets of the same size. A work-around to this is to appropriately pad the input set such that the elements to sort are at the top of the output list. Layer and Pfleiderer [2004] proposed an architecture that can support several input sizes, up to a fixed maximum. Reconfigurable logic is used to change the required shuffles depending on the input width. Another approach, presented in Zhang and Zheng [2000], uses a pipelined sorting network

of fixed input set size m and a set of m queues that each can store n/m elements. Control logic is required to appropriately feed the sorting network.

Sorting networks with better asymptotic cost have been proposed. In particular, Ajtai et al. [1983] present the AKS sorting network, which has the optimal asymptotic cost of $O(n \log n)$. Unfortunately, this result is of purely theoretical value, as the large hidden constant makes these networks unusable in practice.

Another sorting network that has been popular because of its high regularity and simplicity is the odd-even transposition sorting network [Knuth 1968], which, however, requires $O(n)$ stages and $O(n^2)$ comparison operations. As in Stone [1971], it can be folded to reuse only one stage $O(n)$ times.

In summary, odd-even and bitonic sorting networks remain the most popular solutions because of their simplicity and close-to-optimal cost and performance. In this article, we build on bitonic sorters, showing that their inherent regularity can be exploited to create a large novel set of sorters with reduced asymptotic and actual area-cost requirements and a rich set of relevant cost/time trade-offs.

Linear sorters. Linear sorters are linear sequences of n nodes that sort n values, where n is not limited to be a power of two. Each node is composed of a flip-flop and a comparison operator [Perez-Andrade et al. 2009; Lee and Tsai 1995]. The asymptotic time and logic cost of linear sorters is $O(n)$, and they are fully streaming with streaming width 1.

A linear sorter that scales better to larger input set sizes is presented by Lin and Liu [2002]. They propose an expandable architecture that creates a linear array that can store m elements and is able to sort n elements in n/m passes. Another attempt to increase the performance of linear sorters is the interleaved linear sorter (ILS) in Ortiz and Andrews [2010]. It is composed of w linear sorters working in parallel to generate the output list at a rate of w words per cycle by appropriately interleaving the output of the w individual sorters. Ortiz and Andrews show that in practice this architecture does not scale well to larger w values due to the complexity of the interleaving logic.

Table I and Section 5 show how our designs greatly improve over linear sorters.

Other approaches. Other types of hardware sorters achieve better execution times by increasing the amount of resources. Examples of this include 2D and 3D meshes. One of the most popular is the shear-sort algorithm [Scherson and Sen 1989]. It can sort in $O(n \log n)$ time using $O(n)$ nodes, where each node requires a basic processor that performs comparison operations and manages the communication with other nodes. Shear-sort is based on a $\sqrt{n} \times \sqrt{n}$ mesh that sorts and stores n elements. The sorting is performed in $O(\log n)$ steps. In each step, columns and rows are alternately sorted using the folded odd-even transposition sorting network. Additionally, more complex mesh architectures have been shown to sort in constant time [Jang and Prasanna 1992; Chen and Chen 1994]. Lastly, Kutylowsky et al. [2000] describe periodic sorting networks, which apply a form of folding to multidimensional meshes.

These techniques require more functionality to be added to the nodes and more complex communication links and are thus not competitive with linear sorters and sorting networks.

1.2. Our Contributions and Approach

We summarize our contributions and approach as follows.

A flexible class of streaming sorting networks. We present a class of hardware structures, called streaming sorting networks, which yields novel, and in part better, asymptotic cost-performance trade-offs. In particular, the new structures allow for any streaming width $2 \leq w \leq n$, where w divides n, and they can be fully streaming (for high throughput) or non-fully streaming (for low area cost) designs. In the latter case, if $w = 2$, the area cost can be as low as $O(1)$ logic and $O(n)$ storage. The flexibility provided by our design space gives a large benefit by allowing the designer to choose the particular structure that best fits application-specific goals and requirements. Table I compares the cost to prior work. The new streaming networks are derived from different types of bitonic sorting networks by folding (reusing logic). The main challenge is in folding the permutations between stages; for that, we use the recent method from [Püschel et al. 2009; 2012],

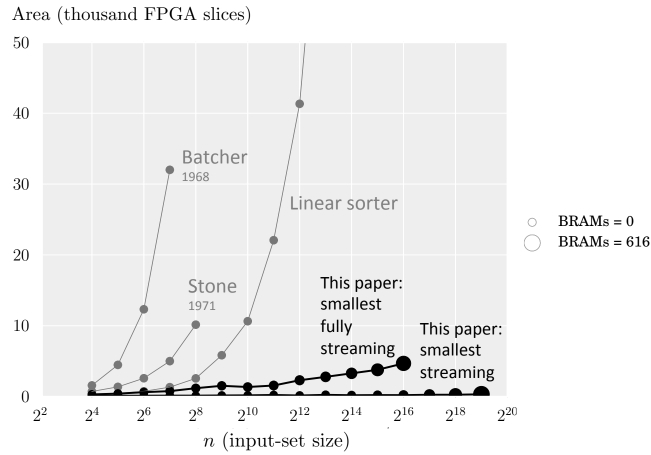


Fig. 2. Area growth trends for some of the sorting networks generated with the techniques proposed in this article in comparison with previous implementation techniques. The largest linear sorter for $n = 2^{13}$, which has been left out of the plot, required 82,000 FPGA slices.

which yields optimal or close-to-optimal solutions using RAM blocks as storage. Our derivation of the streaming networks uses a formal framework, or mathematical domain-specific language (DSL), that we introduce. Using this framework, we give a detailed, and also nonasymptotic, analysis of throughput, latency, and cost of the networks.

Hardware synthesis tool and real cost analysis. Our class of streaming networks offers a rich and novel set of relevant area-performance trade-offs. However, the large size of the class and the complexity of its design structures make manual implementation impractical. As the second main contribution, we create a synthesis tool that generates hardware implementations in RTL Verilog from a description in our mathematical DSL. The resulting code can be synthesized for field-programmable gate-arrays (FPGAs) or ASICs and allows us to explore the entire design space and perform a real area-cost comparison of our networks with prior work on actual hardware. An easy-to-use interface to the synthesis tool is available at Zuluaga et al. [2012b].

As a preview, we show two experiments. Figure 2 shows the cost of implementing various sorters with 16-bit fixed point input values that fit on a Xilinx Virtex-6 FPGA. The x-axis indicates the input size n, the y-axis the number of FPGA configurable slices used, and the size of the marker quantifies the number of BRAMs used (BRAMs are blocks of on-chip memory available in FPGAs). The implementations using Batcher's and Stone's architectures can only sort up to 128 or 256 elements, respectively, on this FPGA. Conversely, our streaming sorting networks with streaming width $w = 2$ can sort up to 2^{19} elements on this FPGA, and our smallest fully streaming design can sort up to 2^{16} elements.

Figure 3 shows all the 256-element sorting networks that we generate with our framework (using 16-bits per element) that fit onto the Virtex-6 FPGA. The x-axis indicates the number of configurable FPGA slices used, the y-axis the maximum achievable throughput in giga samples per second, and the size of the marker indicates the number of BRAMs used. This plot shows that we can generate a wide range of design trade-offs that outperform previous implementations, such as that of Stone and the linear sorter (Batcher's is omitted due to the high cost). For practical applications, only the Pareto-optimal ones (those towards the top left), would be considered.

This article is a considerable extension of the work presented in Zuluaga et al. [2012a]. In particular, we include a quantitative analysis of the cost and performance of the generated designs, a thorough description of the datapath reuse techniques applied to sorting networks, and a detailed analysis on how the complexity of sorting networks can be modified to scale the original algorithms to larger input set sizes. We note that recently, Chen et al. [2015] took an approach similar to Zulu-

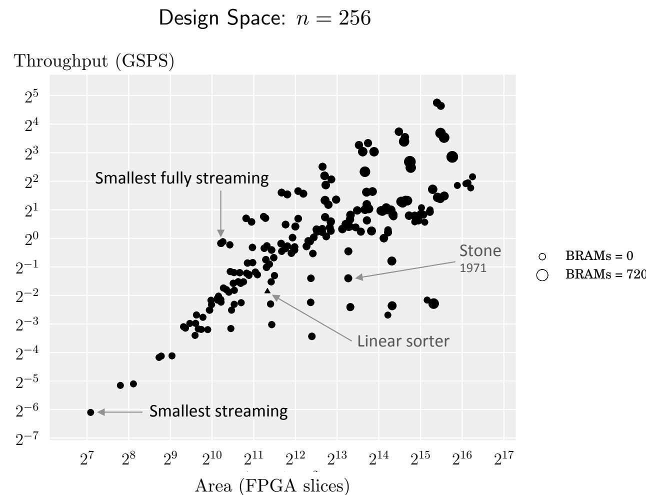


Fig. 3. Design space of generated implementations for $n = 256$ and the linear sorter implementation. The unfolded sorting network originally proposed by Batcher does not fit on the chosen FPGA and thus is not shown in the plot.

aga et al. [2012a] but the designs were implemented by hand (in contrast to being generated), only a portion of our design space was covered, and the method for permuting data (see Section 3.2) was different, requiring fewer words of memory at the cost of higher control complexity.

1.3. Organization

In Section 2, we introduce the mathematical formalism (we will refer to it as DSL) that we use to express sorting network datapaths. We use it to express different variants of bitonic sorting networks, which are the starting point of our work. Section 3 introduces the concept of datapath reuse and shows how it is captured and specified in our DSL using high-level parameters. Application to bitonic sorting yields the large set of streaming networks that are the contribution of this article. We develop cost and performance models for these networks as functions of the parameters that specify the degree of reuse. Section 4 precisely analyzes the different cost-performance trade-offs that can be obtained and shows their asymptotic behavior. In Section 5, to confirm this analysis with real hardware, we present in a tool that generates RTL Verilog from our mathematical DSL. Using this tool, we show a set of experiments and benchmarks against prior work on current FPGAs.

2. BITONIC SORTING NETWORKS

In this section, we provide background on bitonic sorting networks, which are the starting point of our work. We represent these networks using a mathematical formalism that we introduce. This formalism expresses the structure and the components from which sorting networks are built, and it allows the formal manipulation of these networks. Further, as we will explain later, we have implemented this formalism as DSL and compiler to enable the automatic generation of actual hardware implementations of the entire class of sorting networks that we derive in this work to enable careful evaluation. For this reason, we will refer to the formalism as DSL throughout the article. The DSL borrows concepts from Van Loan [1992], Johnson et al. [1990], Franchetti et al. [2009], and Milder et al. [2012].

2.1. A DSL for Sorting Networks

Our DSL is designed to capture the structured dataflow of sorting networks. We first explain the basic principles underlying the language. Then we explain its components.

Basic principles. The elements of the language are operators that map an input array to an output array of the same fixed length. Each operator is either a first-order operator or a composition using higher-order operators. Arrays are written as $x = (x_i)_{0\leq i<n} = (x_0, \ldots, x_{n-1})$, and operators A_n are functions that map an array of length n. For instance, the operator S_2 introduced earlier maps an array of two elements into an ordered array of the same elements. We write $y = A_n x$ to indicate that A_n maps x to y.

First-order operators. We define the following first-order operators that are the basic components needed to represent sorting networks:

$$
\begin{aligned}
I_n &: x \mapsto x\\
J_n &: (x_i)_{0\leq i<n} \mapsto (x_{n-1-i})_{0\leq i<n}\\
L_n^m &: (x_{ik+j})_{\substack{0\leq i<m\\0\leq j<k}} \mapsto (x_{jm+i})_{\substack{0\leq i<m\\0\leq j<k}}; \quad n = km\\
S_2 &: (x_0, x_1) \mapsto (\min(x_0, x_1), \max(x_0, x_1))\\
X_2^c &: \begin{cases} I_2, & \text{for } c = 0\\ J_2 \circ S_2, & \text{for } c = 1\\ S_2, & \text{for } c = 2. \end{cases}
\end{aligned}
$$

I_n is the identity operator, and J_n and L_n^m represent permutations. J_n reverses the input array, and L_n^m performs a stride-by-m permutation on n elements. For instance, $L_n^{n/2}$ is known as the perfect shuffle permutation, and $L_{n^2}^n$ is known as the corner turn or transposition of an $n \times n$ matrix stored linearized in memory.

S_2 and X_2^c are the basic building blocks for sorting networks: S_2 sorts two elements into ascending order, whereas X_2^c (where c is a parameter) can be configured to perform ascending sorting, descending sorting, or to preserve the original order of the input elements.

Higher-order operators. The purpose of higher-order operators is to recursively compose operators into more complex dataflow structures. We define the following high order operators:

— Composition ($\circ$): $A_m \circ B_m$ is the composition of operators, as shown in Figure 4(a). The input array is first mapped by B and then by A. The symbol $\circ$ may be omitted to simplify expressions. When a sequence of operators is used to map the input array, we can use the iterative composition, which is written using the product sign:

$$\prod_{i=0}^{t-1} A_m^{(i)} = A_m^{(0)} \circ \cdots \circ A_m^{(t-1)}.$$

Since composition is applied from right to left, we will draw dataflow graphs also from right to left.

— Direct sum ($\oplus$): $A_m \oplus B_n$ signifies that A_m maps the upper m elements and B_n the bottom n elements of the input array, as shown in an example in Figure 4(b).

— Tensor product ($\otimes$): The expression $I_n \otimes A_m = A_m \oplus \cdots \oplus A_m$ replicates the operator A_m in parallel n times to operate on the input array. An example is shown in Figure 4(c).

— Indexed tensor product ($\otimes_k$): The expression $I_n \otimes_k A_m^{(k)} = A_m^{(0)} \oplus \cdots \oplus A_m^{(n-1)}$ allows for a change in the replicated operator through the parameter k and is represented as in Figure 4(d). All $A_m^{(i)}$ are assumed to have the same size.

2.2. Bitonic Merge

The foundation of bitonic sorting is the merging of bitonic sequences [Batcher 1968]. A regular bitonic sequence is a concatenation of an ascending sequence and a descending sequence, each of size $n/2$. We define M_n as the bitonic merge operator that transforms a regular bitonic sequence of size n into a sorted sequence of the same size. In the case of $n = 2$, $M_2 = S_2$.

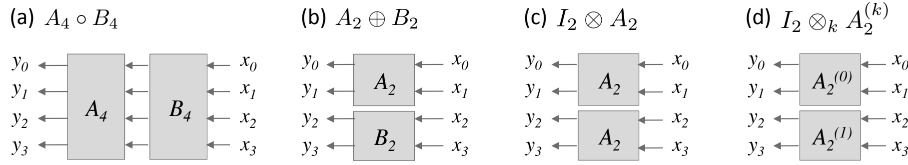


Fig. 4. Higher-order operators and associated dataflow graph structures (from right to left): composition (a), direct sum (b), tensor product (c), and indexed tensor product (d).

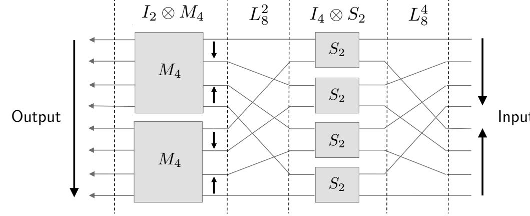


Fig. 5. M_8: Bitonic merger for $n = 8$. The arrows indicate ascending or descending sequences within the data array.

Figure 5 illustrates how to recursively construct bitonic mergers. Given a regular bitonic sequence of size n, performing interleaved pairwise sorting between the two ordered subsequences and a posterior interleaved split generates two regular bitonic sequences of size $n/2$ that can be merged recursively.

The structure in Figure 5 is expressed in our DSL as

$$M_8 = (I_2 \otimes M_4)L_8^2(I_4 \otimes S_2)L_8^4. \tag{1}$$

The merger for a generic input size $n = 2^t$ is analogous [Batcher 1968] and is expressed as

$$M_{2^t} = (I_2 \otimes M_{2^{t-1}})L_{2^t}^2(I_{2^{t-1}} \otimes S_2)L_{2^t}^{2^{t-1}}. \tag{2}$$

Recursive expansion of (2) and proper parenthesizing yields a complete decomposition into basic operators:

$$M_{2^t} = \prod_{j=1}^{t} \left[(I_{2^{t-j}} \otimes L_{2^j}^2)(I_{2^{t-1}} \otimes S_2)(I_{2^{t-j}} \otimes L_{2^j}^{2^{j-1}})\right]. \tag{3}$$

This expression shows that the merging of bitonic sequences can be done in $t = \log_2 n$ stages. Each stage consists of $n/2$ parallel S_2 blocks. The stages are connected by permutations that change from stage to stage (since they depend on j). Similar to how the Pease FFT [Pease 1968] is obtained, this expression can be manipulated using tensor product identities [Johnson et al. 1990] into the "constant geometry" form:

$$M_{2^t} = \prod_{j=1}^{t} \left[(I_{2^{t-1}} \otimes S_2)L_{2^t}^{2^{t-1}}\right]. \tag{4}$$

The permutation is now the same in each iteration. This manipulation is also applied to sorting networks in Layer and Pfleiderer [2004].

2.3. Bitonic Sort

Bitonic sorting networks iteratively merge regular bitonic sequences, as illustrated in Figure 6. The classical bitonic sorting network consists of a sequence of $\log_2 n$ merging stages following the *sorting by merging* principle [Batcher 1968]. In the first stage, input pairs are merged to form sorted lists of two elements; in the final stage, two lists of size $n/2$ are merged. After every merging

stage, half of the sorted lists are inverted to create a bitonic sequence at the input of every merger. The vertical arrows in Figure 6 again show sorted subsequences (ascending or descending) at each stage.

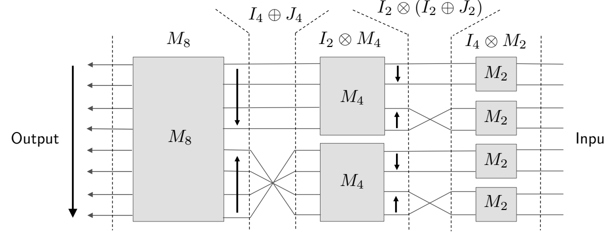


Fig. 6. Bitonic sorter using bitonic mergers for $n = 8$.

We define the sorting operator S_n, which transforms an input array of size n into a sorted ascending sequence of the same size. From Figure 6 we can express S_8 as

$$S_8 = M_8(I_4 \oplus J_4)(I_2 \otimes M_4)(I_2 \otimes (I_2 \oplus J_2))(I_4 \otimes M_2). \tag{5}$$

The generic expression for S_{2^t} is analogous:

$$S_{2^t} = \prod_{i=t}^{1} [(I_{2^{t-i}} \otimes M_{2^i})(I_{2^{t-i}} \otimes (I_{2^{i-1}} \otimes J_{2^{i-1}}))]. \tag{6}$$

This expression shows that bitonic sorting consists of a sequence of $t = \log_2 n$ merging stages.

2.4. Sorting Networks as Breakdown Rules

Next, we use our DSL to represent several variants of bitonic sorting networks as *breakdown rules*. A breakdown rule is an expression like (6) that decomposes the sorting operator S_n into basic operators. Five variants of bitonic sorting networks have been derived from the literature. To express them as rules, we first define the following permutations:

$$P_{2^t}^{2^j} = I_{2^{t-j}} \otimes (I_2 \otimes L_{2^{j-1}}^{2^{j-2}})L_{2^j}^2 \tag{7}$$

$$R_{2^t}^{2^i} = I_{2^{i-1}} \otimes L_{2^{t-i+1}}^{2^{t-i}} \tag{8}$$

$$Q_{2^t}^{2^i} = I_{2^{i-1}} \otimes (I_{2^{t-i}} \oplus J_{2^{t-i}}) \tag{9}$$

We denote the five sorting networks by SN1–5; each one can be derived by applying suitable transformations to (6):

— SN1 is derived by inserting (3) into (6) [Batcher 1968].
— SN2 is derived by inserting (4) into (6) [Layer and Pfleiderer 2004]. By using (4), the regularity of the design increases, but it also increases the complexity of the permutations occurring in the merging stages.
— SN3 is obtained by rewriting SN1 to use the operator X_2^c instead of S_2 to eliminate the J_n permutations [Batcher 1968]. Therefore, the flips are incorporated into the pairwise ordering process. SN3 represents the trade-off of eliminating a nontrivial permutation at the cost of the additional control logic for X_2^c specified by the function $g(i, m)$.
— SN4 is obtained similarly to SN3, namely by rewriting SN2 to use X_2^c, thus eliminating the permutations J_n [Layer and Pfleiderer 2004].
— As for SN5, it has been demonstrated by Stone [1971] that a bitonic sorting network can be implemented in t^2 identical stages consisting of parallel pairwise comparisons followed by the perfect shuffle permutation. Each of the prior networks has $t(t + 1)/2$ stages with different permutations

Table II. Bitonic Sorting Networks as Breakdown Rules for S_n

SN1:	$\prod_{i=1}^{t-1}\left[(I_{2^{t-1}}\otimes S_2)\prod_{j=2}^{t-i+1}\left[P_{2^t}^{2^j}(I_{2^{t-1}}\otimes S_2)\right]R_{2^t}^{2^i}Q_{2^t}^{2^i}\right](I_{2^{t-1}}\otimes S_2)$
SN2:	$\prod_{i=1}^{t-1}\left[\prod_{j=1}^{t-i+1}\left[(I_{2^{t-1}}\otimes S_2)R_{2^t}^{2^i}\right]Q_{2^t}^{2^i}\right](I_{2^{t-1}}\otimes S_2)$
SN3:	$\prod_{i=1}^{t-1}\left[(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})\prod_{j=2}^{t-i+1}\left[P_{2^t}^{2^j}(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})\right]R_{2^t}^{2^i}\right](I_{2^{t-1}}\otimes_m X_2^{g(i,m)})$; $g(i,m)=\begin{cases}1, & m[t-i]=1 \text{ and } i\neq 1\\ 2, & m[t-i]=0 \text{ or } i=1\end{cases}$
SN4:	$\prod_{i=1}^{t-1}\left[\prod_{j=1}^{t-i+1}\left[(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})R_{2^t}^{2^i}\right]\right](I_{2^{t-1}}\otimes_m X_2^{g(t,m)})$
SN5:	$\prod_{i=0}^{t-1}\prod_{j=0}^{t-1}\left[(I_{2^{t-1}}\otimes_m X_2^{f(i,j,m)})L_{2^t}^{2^{t-1}}\right]$; $f(i,j,m)=\begin{cases}0, & t-1<j+i\\ 1, & m[t-1-j-i]=1 \text{ and } i\neq 0\\ 2, & m[t-1-j-i]=0 \text{ or } i=0\end{cases}$

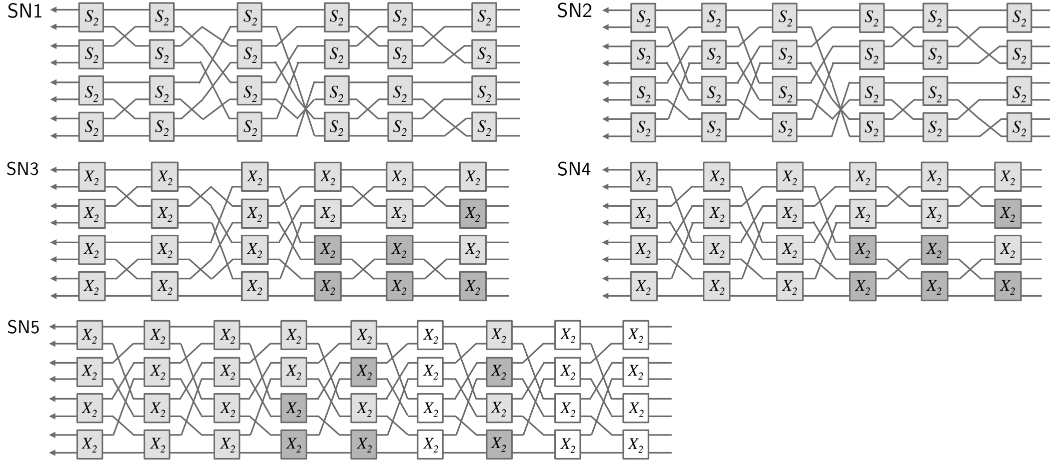


Fig. 7. Dataflow graph of S_8 using each of our five breakdown rules. The configuration of X_2 is shown by the shading: white ($c=0$), dark gray ($c=1$), light gray (c=2).

in each stage. Thus, SN5 increases the number of stages of the computation in exchange for the perfect regularity of the permutation stages. SN5 requires the configurable sorters X_2^c.

Table II shows the DSL breakdown rules for each sorting network; Figure 7 shows the associated dataflow of each for the example S_8.

A high-level cost and performance analysis of the five sorting networks from Table II is straightforward. Each is fully streaming with width $n=2^t$ (i.e., has a throughput of n words per cycle). SN1 through 4 use $t(t+1)/2$ stages for a total cost of $t(t+1)2^{t-2}$ 2-input sorters. The more regular SN1 uses t^2 stages for a total cost of $t^2 2^{t-1}$ 2-input sorters.

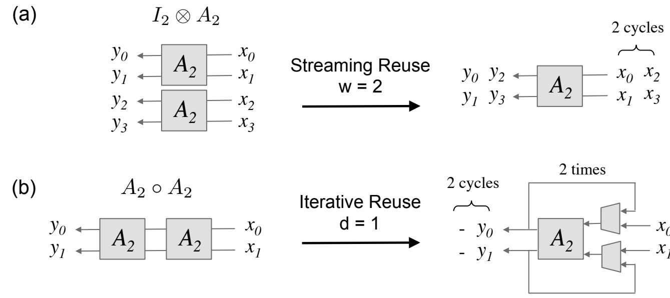


Fig. 8. Examples of streaming reuse (a) and iterative reuse (b).

3. REUSE IN BITONIC SORTING NETWORKS

Given the obvious regularity of bitonic sorting networks, there are opportunities to reuse datapath elements. In other words, sorting networks can be "folded" in various ways to *reuse* hardware blocks such as S_2 or X_2^c by time multiplexing them. This way, the cost can be decreased at the price of an increased time to perform sorting. Different degrees of folding yield different cost/performance trade-offs. The challenge in folding lies in handling the permutations between stages, which is likely the reason that folding was done previously only in very limited ways.

In this section, we first introduce the principles of datapath reuse and describe how we represent them in our DSL. We discuss the reuse opportunities in each of the breakdown rules for sorting networks introduced in the previous section, and then we execute the folding by streaming the occurring permutations. This folding will incur the need for additional logic and memory. We provide a detailed cost and performance analysis of the folded networks as a function of the degree of reuse.

3.1. DSL Implementation Directives to Express Reuse

The formal language that we use to represent dataflow graphs clearly identifies two types of reuse opportunities. For each type, we define an *implementation directive* that specifies the desired degree of reuse in the DSL.

Streaming reuse. The expression $I_2 \otimes A_2$ represents a dataflow graph in which two modules A_2 are applied in parallel to the input array, as shown in the left-hand side of Figure 8(a). The same computation can be performed by instantiating only one operator A_2 and time multiplexing it. First, it operates on the top two elements of the input array, and then (in the next clock cycle) on the remaining two, as shown in the right-hand side of Figure 8(a). We refer to this type of reuse as *streaming reuse*, as it leads to streaming designs (Definition 1.1).

In a general form, streaming reuse can be naturally applied to expressions of the form $I_p \otimes A_m$, as illustrated in Figure 9(a). The degree of freedom in this type of reuse, and thus our implementation directive, is the *streaming width* $w = sm$, where p must be evenly divisible by s. This means that s parallel blocks of A_m are instantiated and reused p/s times. Accordingly, the input has to be divided into parts of length w that are fed in p/s consecutive cycles. The minimum valid streaming width of $I_p \otimes A_m$ is m, which leads to the smallest design where only a single A_m module is implemented. Greater values of w result in better performance but incur higher implementation costs.

Streaming reuse can also be applied to indexed tensor products of the form $I_p \otimes_k A_m^{(k)}$, provided that the operator $A_m^{(k)}$ can be configured through a suitable control to perform the functionality required for any value of k. X_2^c is an example of such a configurable operator.

Iterative reuse. Next, consider the expression $A_2 \circ A_2$ that represents a dataflow graph in which two modules A_2 are applied sequentially to the input array. This is shown in the left-hand side of Figure 8(b). In this case, time multiplexing is performed by building a single instance of operator A_2 and allowing data to feed back from its output to its input, as shown in the right-hand side of

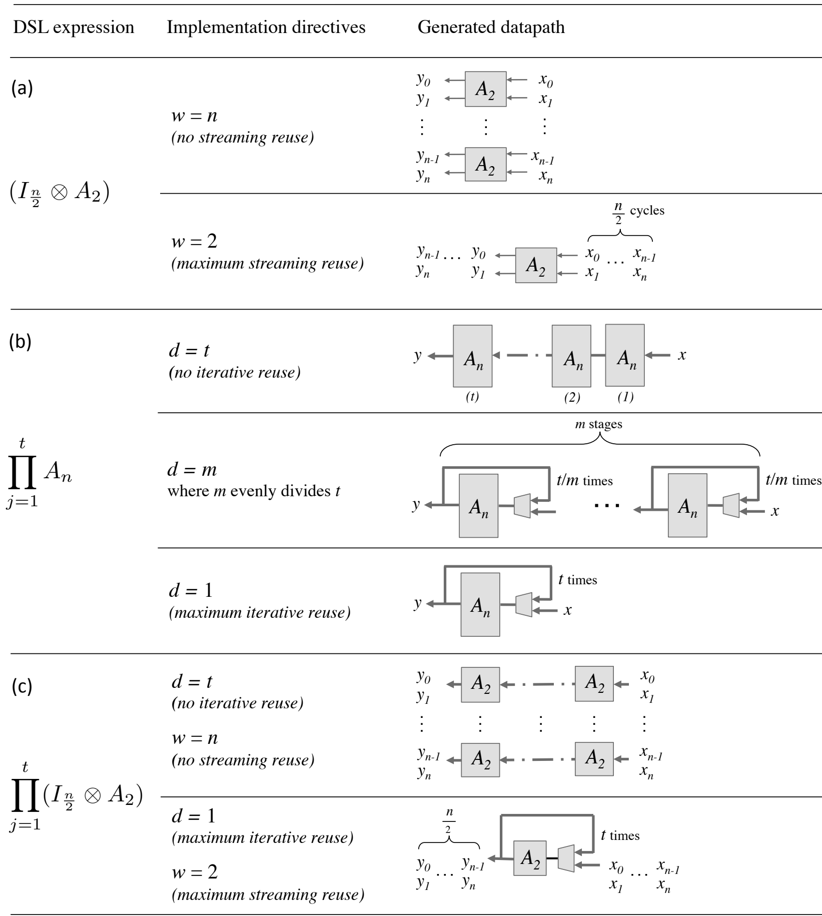


Fig. 9. (a) Streaming reuse. (b) Iterative reuse. (c) Iterative and streaming reuse.

Figure 8(b). As a consequence, a second input array can only be fed to the system after a delay. We refer to this type of reuse as *iterative reuse*.

Definition 3.1 (*Iterative Designs*). When iterative reuse is applied, we refer to the resulting design as iterative. Iterative designs must halt the input stream until the input set recirculates around the reused block the designated number of times. Designs that are iterative are not fully streaming (Definition 1.2).

In a general form, iterative reuse can be naturally applied to the iterative composition operator $\prod_{j=1}^{t} A_n$, as shown in Figure 9(b). The degree of freedom of this type of reuse, and thus our implementation directive, is the *depth* d. We say that the expression $\prod_{j=1}^{t} A_n$ is implemented with depth $d \mid t$, if d operators A_n are built and reused t/d times. If $d < t$, the resulting design is iterative. If $d = t$ the design is fully streaming. The implementation with minimum area and minimum throughput is achieved when $d = 1$.

Iterative reuse and streaming reuse can be combined in more complex DSL expressions, as shown in Figure 9(c).

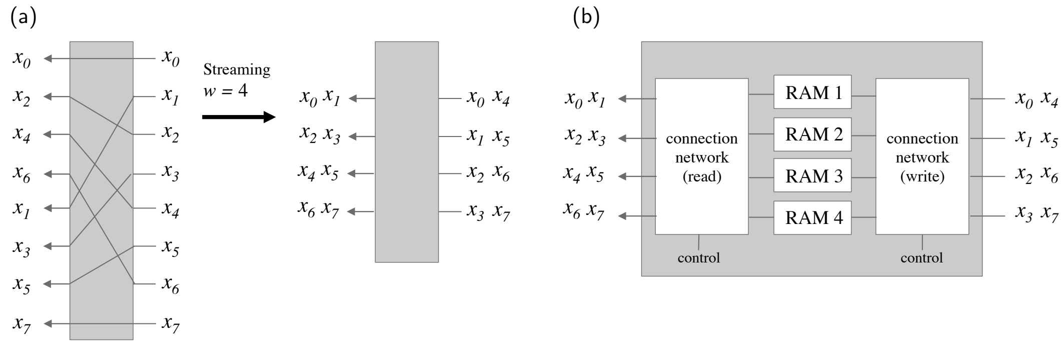


Fig. 10. (a) An example permutation on eight points and streamed with $w = 4$. (b) Hardware architecture to stream the permutation in (a) as proposed in Püschel et al. [2009].

3.2. Applying Streaming Reuse to Permutations

It is easy to see how streaming reuse and iterative reuse can be applied to many of the components of our DSL. However, one remaining difficulty is to apply streaming reuse to the ubiquitous permutations $P_{2^t}^{2^j}$, $R_{2^t}^{2^i}$, and $Q_{2^t}^{2^i}$. To illustrate the difficulty, Figure 10(a) shows an example of the desired behavior of a streaming permutation P_8 with streaming width $w = 4$. On the left-hand side, we see the direct version, which reorders eight data elements in space. However, if we apply streaming reuse with $w = 4$, then our inputs are received four at a time, and they are reordered *across two clock cycles* (i.e., not only in space but also in time, thus requiring memory).

Püschel et al. [2009; 2012] proposed a method to build a hardware structure that performs a fixed permutation on streaming data. This method is applicable to a (clearly defined) subset of all permutations that includes the ones needed here. For a permutation on n points with streaming width w, the resulting architecture uses w independent double ported memories, each with a capacity of $2n/w$ words, and two interconnection networks, as illustrated in Figure 10(b). In each cycle, one word is written to each memory and one word is read from each memory. Püschel et al. [2009] provide algorithms to efficiently find the two interconnection networks and the associated control logic required to prevent any bank conflict in accessing the RAMs. The resulting design is fully streaming. Appendix A derives these solutions for the permutations needed in SN1–5.

3.3. Reuse in Breakdown Rules

Each breakdown rule SN1 through 5 offers different opportunities for reuse. The DSL representation reveals these options (i.e., whether iterative or streaming reuse or both are applicable). Table 3.3 shows the result, including the possible choices of depth and streaming width in each case. This table is discussed next.

Streaming reuse. Streaming reuse with streaming width w can be applied to an expression when all of its components can be streamed with w. Components in bitonic sorting networks are either permutations or expressions of the form $(I_{2^{t-1}} \otimes S_2)$ or $(I_{2^{t-1}} \otimes_c X_2^c)$, which can be streamed with any streaming width $w \mid 2^t$, $w \geq 2$.

Further, Appendix A shows that the permutations $P_{2^t}^{2^j}$, $R_{2^t}^{2^i}$, and $Q_{2^t}^{2^i}$ in (7), (8) and (9) can be streamed using the techniques in Püschel et al. [2009]. In summary, all five sorting networks can be streamed with any streaming width $w \mid 2^t$, $w \geq 2$.

Iterative reuse. Iterative reuse is applicable to iterative composition operators $\prod_i$ with a constant computational body. Constant means that the body does not change with i *or* that its only dependence on i is in a control function such as $g(i, m)$ in Table II. For this reason, iterative reuse cannot be applied to SN1 and SN3, because the body of outer compositions depends on i and the body of the inner composition on j. However, iterative reuse can be partially applied to SN2 and SN4, namely to the inner iterative composition whose body does not depend on the iteration index

Table III. Implementation Characteristics of the Different Breakdown Rules When Applying Streaming and Iterative Reuse

Breakdown rule	Implementation Directives		
	Degree of Reuse	Ranges	Constraints
SN1, SN3	w	$2 \leq w \leq n$	$w \mid n$
SN2, SN4	w, d_i	$1 \leq i \leq t-1, \quad 1 \leq d_i \leq t-i+1$ $2 \leq w \leq n$	$d_i \mid t-i+1$ $w \mid n$
SN5	w, d, d'	$1 \leq d, d' \leq t$ $2 \leq w \leq n$	$d \mid t, d' \mid t$ $w \mid n$

Note: The constraint $a \mid b$ means that a must divide b evenly.

j. In SN5, iterative reuse can be applied to both iterative compositions, as the computational body does not depend on any of the iteration indices. Thus, the smallest such implementation of SN5 is a single physical stage that is reused t^2 times, as was proposed by Stone [1971].

Example. We consider the possible implementations of S_8 ($n = 8$, $t = \log_2 n = 3$) using the different breakdown rules and degrees of reuse.

For all breakdown rules, valid streaming widths w are 2, 4, and 8. This is the only choice available for SN1 and SN3. For SN2 and SN4, in addition the depths d_i, $i = 1, 2$, need to be specified. Using Table 3.3, possible values for d_1 are 1 and 3, and possible values for d_2 are 1 and 2. Maximum iterative reuse is obtained with $d_1 = 1$ and $d_2 = 1$. For SN5, the depths d and d' need to be specified. Using Table 3.3, possible values for d and d' are 1 and 3. Maximum iterative reuse is obtained with $d = 1$ and $d' = 1$. The entire set of choices (consisting of five breakdown rules combined with their implementation directives) yields a design space of 39 different implementations for S_8. The size of the design space becomes larger as n grows. For instance, S_{16} has 93 different implementations, S_{32} has 159, and S_{64} has 603.

3.4. Cost Analysis

In this section, we provide a detailed cost analysis of the considered streaming sorting networks with different streaming and iterative reuse. Using this analysis, we can reason about the design space and understand the relative strengths and weaknesses of the different options. Later, in Section 5 we will also experimentally measure the cost of many designs on FPGAs.

The implementation cost of a sorting network with reuse is determined by the choice of network (SN1 through 5) and the implementation directives (streaming width w and depth values d, as shown in Table 3.3). All implementations require comparison operations, whose overall cost is proportional to the number of S_2 and X_2 operators used. Additional cost is incurred when the sorting network is streamed (i.e. when $w < n$). In this case, every permutation is implemented using w memory banks and two switching networks. Thus, we measure the cost of the streaming permutations by the number of 2-input switches required to build the two switching networks and the number of memory words required to build the w banks. We assume that the cost of control logic required to configure the switching networks and to read and write from the memory banks is negligible. This assumption is confirmed by our experiments.

Comparison operations. Table IV (second column) shows the number of 2-input sorters (S_2 or X_2) required for the implementation of each breakdown rule as a function of the streaming width w and the depths d, when applicable. One can determine the number of 2-input sorters in a straightforward manner by using the DSL expressions and specified directives.

Switches and RAM words. Streaming permutations are constructed using switches and RAMs. Counting them is more complex, because as explained by Püschel et al. [2009], the exact cost depends on the the exact permutation and streaming width w, as detailed in Appendix B. The result is shown in the last two columns of Table IV.

Table IV. Number of Basic Components Required for S_{2^t}, Built with Streaming Reuse ($w = 2^k$) and Iterative Reuse (d Parameters)

Breakdown rule	2-input sorters	Switches	RAM words
SN1	$2^{k-2}t(t+1)$	if $(k=1)$: $(t-1)(t+1)$ else: $2^{k-1}(t-k)(t-k+3)$	$4(2^k(k-t-3)+2^t3)$
SN2	$2^{k-1}\left(1+\sum_{i=1}^{t-1} d_i\right)$	if $((d_i = t-i+1)_{1\le i\le t-1})$: if $(k=1)$: $2^{k-1}t^2$ else: $2^{k-1}(t^2-k^2+t-k)$ else: $2^k\left(\left(\sum_{i=1}^{t-k} d_i\right)+t-k\right)$	if $((d_i = t-i+1)_{1\le i\le t-1})$: $4(2^k - 2^k k + 2^t(t-1))$ else: $\sum_{i=1}^{t-k}(d_i+1)2^{t-i+2}$
SN3	$2^{k-2}t(t+1)$	$2^{k-1}(t-k)(t-k+3)$	$4(2^k(k-t-3)+2^t3)$
SN4	$2^{k-1}\left(1+\sum_{i=1}^{t-1} d_i\right)$	$2^k\sum_{i=1}^{t-k} d_i$	$\sum_{i=1}^{t-k} d_i 2^{t-i+2}$
SN5	$2^{k-1}dd'$	if $(k=t)$: 0 else: $2^k dd'$	if $(k=t)$: 0 else: $2^{t+1}dd'$

Note: The cost of SN2 may be less in practice when some but not all d_i are maximal.

Table V. Performance Estimate for a Sorting Network That Processes $n = 2^t$ Elements When Applying Streaming Reuse ($w = 2^k$) and Iterative Reuse (d_i, d, d')

Breakdown Rule	Throughput (Words per Cycle)
SN1, SN3	w
SN2, SN4	if $(d_i = t-i+1)_{1\le i\le t-1}$: w else: $\left(\frac{nw}{n+wc}\right)\min\left(\frac{d_i}{t-i+1}\right)_{1\le i\le t-1}$
SN5	if $(dd' = t^2)$: w else: $\left(\frac{nw}{n+wc}\right)\frac{dd'}{t^2}$

3.5. Performance Analysis

In this section, we analyze the expressions of the five breakdown rules for sorting networks to derive an approximation of the throughput that they can achieve as a function of the input set size ($n = 2^t$) and the implementation directives for streaming and iterative reuse. The results are summarized in Table V.

In the analysis we use the *gap* of a datapath, which is the number of cycles between starts of consecutive input sets. Smaller values of the gap indicate faster processing. The minimal gap is one, which implies that the system begins processing a new input set on every clock cycle. Streaming reuse with width w implies a minimal possible gap of n/w, because it takes n/w cycles to stream one input set of size n into the system.

Fully streaming designs. SN1 and SN3 always generate fully streaming designs because no iterative reuse can be applied to their structures. SN2, SN4, and SN5 produce fully streaming designs only when no iterative reuse is applied (i.e., when the d parameters have their maximum value). These designs will have a gap of n/w and a throughput of w.

Iterative designs. The gap for iterative designs is equal to the latency of the reused block multiplied by the number of times that the block is reused. When iterative reuse is applied on sub-blocks of a larger system, the latency of the reused block has to be at least n/w cycles to allow the entire input set to be received before the data begins recirculating back to the input to begin the second iteration (see Figure 9(b)).

The latency in cycles depends on how it is implemented in RTL (e.g., the granularity of pipelining). To reason about latency we will use an approximation in the following.

The reused block in SN5 is generated from the expression $(I_{2^{t-1}} \otimes_m X_2^{f(i,j,m)})L_{2^t}^{2^{t-1}}$. Because the latency of the permutation $L_{2^t}^{2^{t-1}}$ will be much higher than the latency of the operation $(I_{2^{t-1}} \otimes_m X_2^{f(i,j,m)})$ (which is typically just one cycle), the overall block's latency is dominated by the permutation. Because the cost of a streaming permutation varies with both the permutation and its streaming width, its exact latency is difficult to predict with a simple formula. We can calculate the latency exactly using the method in Püschel et al. [2009]. For simplicity, here we approximate this latency using a simple lower bound and obtain the estimate for SN5 shown in Table V.

The sorting networks SN2 and SN4 each contain $t-1$ separate iterative compositions (the innermost $\prod$ in each expression). One may choose to apply iterative reuse to each of these separate compositions independently by assigning depth values d_i, $1 \leq i < t-i+1$. In these networks, the reused block is of the form $(I_{2^{t-1}} \otimes A_2)R_{2^t}^{2^i}$, where $A_2 = S_2$ and $A_2 = X_2$, respectively. Again, we approximate the latency of these blocks based on the lower bound of the latency of $R_{2^t}^{2^i}$, which is $n/(2^i w)$. Since $n/(2^i w)$ is always smaller than n/w (the minimum latency for iterative reuse), the latency of each block is n/w plus a small implementation-dependent constant c. The gap of the system corresponds to the largest gap found amongst the i blocks, given by

$$\left(\frac{n}{w} + c\right) \max\left(\frac{t-i+1}{d_i}\right)_{1 \leq i \leq t-1}.$$

Given this, we obtain the approximation of the achievable throughput for SN2 and SN4 in Table V.

4. IMPLEMENTATION TRADE-OFFS

As explained in the previous sections, a large set of cost-performance trade-offs can be obtained by considering different breakdown rules with different levels of streaming and iterative reuse. In this section, we use the models developed in Sections 3.4 and 3.5 to draw conclusions about the role of each breakdown rule in the generated design space.

4.1. Optimal and Suboptimal Trade-offs

Given the complexity of the design space, it is not possible to know exactly which choice is best for a particular area budget or performance requirement. However, using the formulas in Tables IV and V, we can draw some preliminary conclusions.

Fully streaming designs. Fully streaming designs can be generated with any of the five breakdown rules and for any given w words per cycle. Therefore, by inspecting the resource requirements from Table IV (with all depths set maximally), we can derive that SN1 generates those with the lowest resource requirements.

Iterative designs. As said before, SN2, SN4, and SN5 are the only candidates for iterative reuse. Table V shows that SN2 and SN4 offer the same throughput, in which case we can compare their resource requirements from Table IV to conclude which one is the best alternative. Because it does not require a permutation between its merging stages, SN4 has lower cost and thus dominates SN2. The number of designs generated by SN2 and SN4 quickly grows with n due to the large number of possible depths d_i, $1 \leq i \leq t-1$. However, closer inspection shows the suboptimality of many of these choices—namely, if it is possible to decrease the value of any d_i while maintaining the same throughput, then the design is suboptimal. In other words, increasing a d_i that does not represent the bottleneck of the system leads to suboptimal designs. Finally, the most regular algorithm SN5 offers additional cost savings through higher iterative reuse at the cost of further decreasing throughput. Thus, SN5 also yields the smallest (but slowest) implementation of a bitonic sorting network.

In summary, when a fully streaming design is required, SN1 is used. On the other hand, when an iterative design is required, SN4 offers an intermediate trade-off between cost and performance, and SN5 provides the absolute lowest cost implementation. In Section 5, we verify this theoretical discussion of trade-offs using synthesized results of these networks.

4.2. Asymptotic Cost and Performance

We analyze the asymptotic cost of our implementations in terms of the required number of logic components and memory words using Table IV and by unifying the number of required switches and 2-input sorters (S_2 or X_2), as these two components are implemented with similar combinatorial logic.

Figure 11 shows a graphical representation of the available implementation trade-offs and indicates their corresponding asymptotic cost and throughput. Figure 11(a) shows the fully unfolded sorting network in which no reuse is applied. This is the fastest implementation with a logic cost of $O(n \log^2 n)$, and no memory required. The throughput of this implementation is n words per cycle, which is the maximum throughput that can be obtained.

Streaming reuse applied to SN1 (Figure 11(b)) reduces throughput to w words per cycle, and at the same time reduces logic cost to $O(w \log^2 n)$, where w is a variable that can take values from 2 to $n/2$. For example, if $w = 2$, the logic cost is reduced to $O(\log^2 n)$. However, memory components are needed for this type of implementation, where the number of words required grows with n at a rate of $O(n \log^2 n)$ independently of w. Therefore, streaming reuse transfers the $O(n)$ component of the cost from logic to memory.

On the other hand, iterative reuse can be applied to SN4 and SN5, as shown in Figure 11(c). By applying iterative reuse to SN4, the implementation cost of $O(n \log^2 n)$ logic components can be reduced to $O(n \log n)$ by choosing the appropriate values for d_i. Similarly, by applying iterative reuse to SN5, the implementation cost of $O(n \log^2 n)$ can be reduced to $O(n)$. This is because SN4 implements a minimum of $\log n$ comparison stages and SN5 implements a minimum of one comparison stage. However, the $O(n)$ component still dominates the logic cost. This type of reuse decreases the throughput by a factor of t for SN4, and by a factor of t^2 for SN5.

Combining these two reuse techniques we obtain designs such as those shown in Fig 11(d). The smallest implementations have a logic cost of $O(\log n)$ and $O(1)$, with a memory cost of $O(n \log n)$ and $O(n)$, respectively. In summary, iterative reuse reduces the $O(\log^2 n)$ logic to $O(\log n)$ or $O(1)$ by reducing throughput by a factor of t or t^2. Streaming reuse can reduce the $O(n)$ logic by reducing the throughput to w words per cycle and by adding memory requirements that grow linearly with n.

This transfer in cost from logic to memory is an important result, as it enables the scaling of sorting networks to larger input sets. To show how this has an impact on the overall cost of the implementation in practice, we can estimate the relative area of n 2-input sorters with registers versus the cost of implementing SRAMs with a capacity of n words. (We include registers with the 2-input sorters because typically pipelining registers are used on the output of the switches.)

First, we estimated the area of these components (assuming 16-bits per word) in the uk65lscllmvbbr_a02 standard cell library for the UMC 65nm technology using Synopsys Design Compiler D-2010.03-SP1-1. Figure 12 (left) shows the estimated area of each of these components. There is an additional overhead included for a memory array due to the logic required to perform read and write operations; however, although this overhead is not negligible for small values of n, for $n > 16$ it is cheaper to use SRAMs than building n individual flip-flops.

The area of n such components was then extrapolated in Figure 12 (right) to create an estimation of the growth in area of n 2-input sorters with registers versus SRAM memories with a capacity of n words. These trends clearly show that the cost of implementing n words in SRAM is much lower than the cost of implementing either n 2-input sorters or n flip-flops. Although the cost of these components might change when using a different technology or data type, these trends are highly likely to show the same behavior.

These area estimates allow us to combine memory and logic area requirements to obtain the cost trends for some of our implementations. Figure 13(a) shows the estimated cost growth for the fully expanded sorting network SN1, considered in Batcher [1968], where no reuse is applied. In contrast, this figure also shows the cost growth when maximum streaming reuse is applied (i.e., when $w = 2$). Figure 13(b) shows the estimated cost growth for the sorting network SN5 with maximal iterative

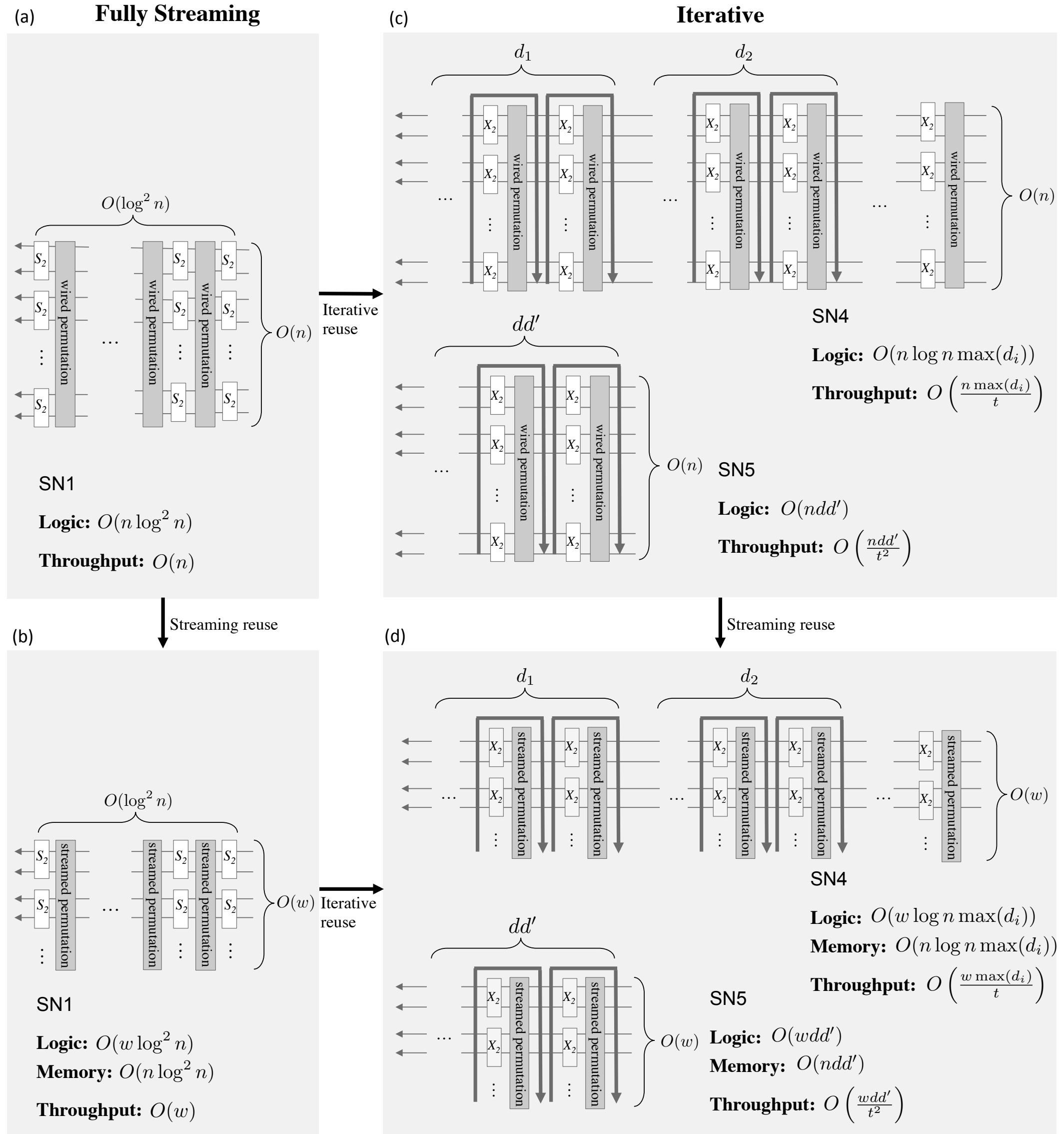


Fig. 11. Implementation trade-offs obtained with the breakdown rules and the reuse variables (streaming width and depths).

reuse; this is the smallest sorting network proposed in previous work [Stone 1971]. In contrast, this figure shows the area growth when maximum streaming reuse additionally is applied.

These trends show that streaming reuse dramatically reduces the cost of sorting network implementations, thus enabling larger input sets to be processed within a given area budget.

4.3. Modeling the Design Space

Using the models developed in Sections 3.4 and 3.5 and the area estimates in Figure 12 (left), we can estimate the performance and cost differences between the possible implementations. In doing so, we can also validate the conclusions drawn in Section 4.1 about the competitiveness of the different breakdown rules across the space of reuse parameters.

Component	Area (μm^2)
2-input sorter	196
1-word in flip-flops	192
1 word in SRAM	32
SRAM overhead	4200

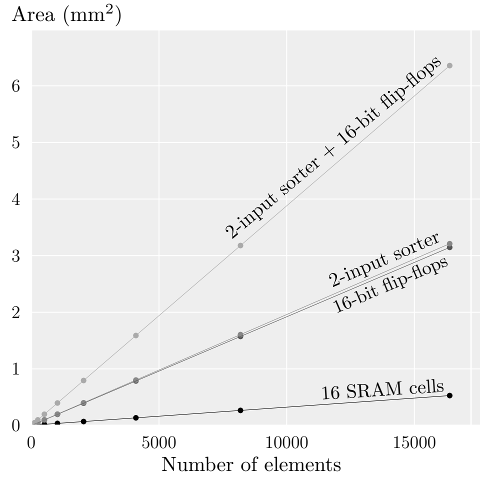


Fig. 12. Left: Estimated area requirements of the basic elements for sorting network implementations with 16-bit fixed point input values. Right: Area cost as a function of number of components required (n).

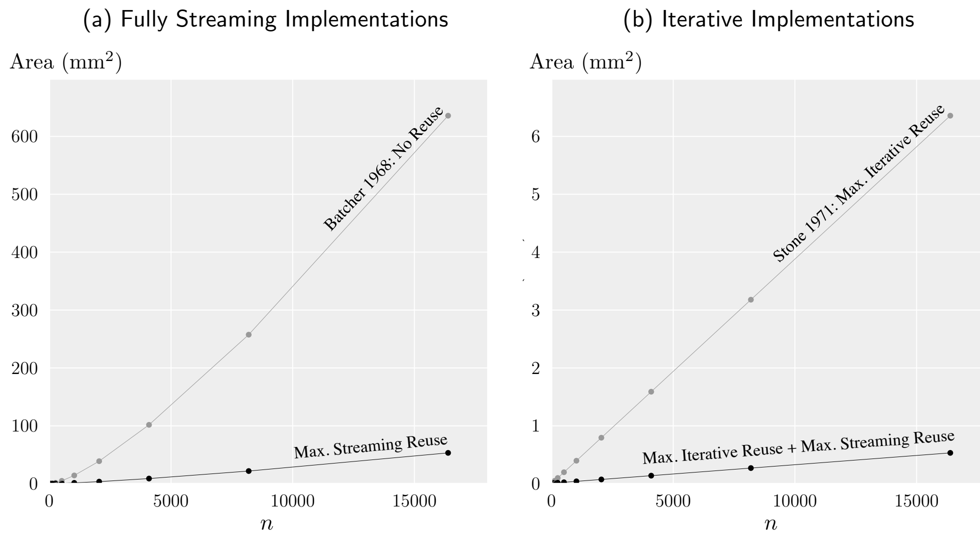


Fig. 13. Estimated area requirement of four different implementation trade-offs.

Figure 14 shows the predicted area and throughput for the designs obtained with $n = 256$ and with $n = 2048$ (assuming 16-bit data words). We constrained the streaming width to a maximum of 32, as designs with larger values become difficult to feed with input data at the requisite high rate. Additionally, the suboptimal d_i combinations mentioned in Section 4.1 were also left out. As the nature of the generated design space is exponential in area and throughput, we use a logarithmic scale to display the obtained range of trade-offs. Circular markers indicate the three breakdown rules that, according to our analysis in Section 4.1, are the candidates for generating the best designs.

As expected, there is no competitive design generated with SN3, while SN1 offers all the Pareto-optimal fully streaming designs with streaming width 2, 4, 8, 16 and 32. Similarly, SN2 does not offer any optimal design, as SN4 generates similar trade-offs at lower cost. On the other hand, SN5 offers the smallest optimal designs but quickly becomes suboptimal when increasing its area

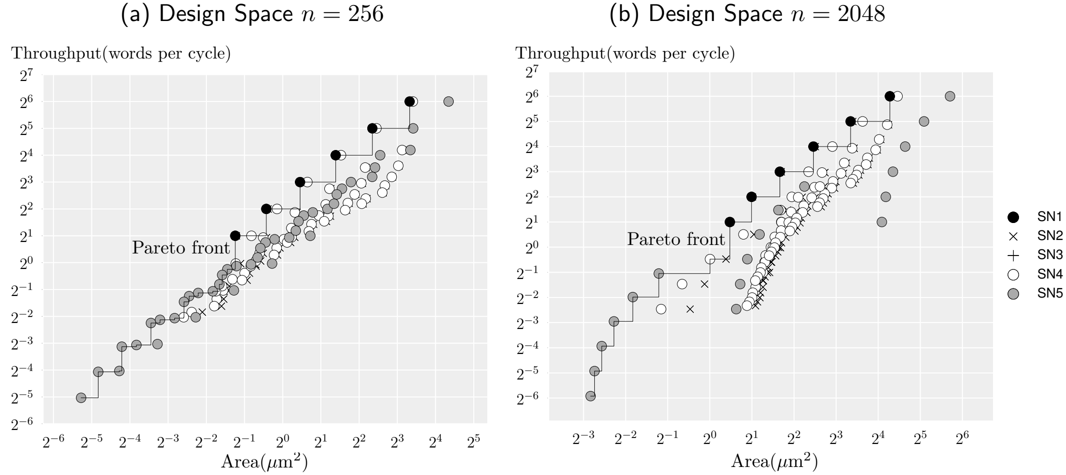


Fig. 14. Estimated area and throughput of the designs generated for S_{256} and S_{2048}.

requirements with larger values of w, d and d'. In other words, the throughput of SN5 does not scale well as resources are increased. These observations hold for both sizes considered. Note that SN5 generates more designs for $n = 256$ than for $n = 2048$. This is because $t = \log_2 n$ has more divisors in this case, which allows for more possible combinations of d and d'.

These models could be also used to quickly predict the Pareto-optimal designs, which are marked in Figure 14. In the following section, we show the actual values of area and throughput that are obtained when generating and synthesizing the designs to target a specific FPGA platform.

5. EXPERIMENTAL EVALUATION

Based on our DSL (Section 2), we have built a tool that can generate for a given n the entire design space of sorting networks derived in this article. Up to now, we have analyzed and modeled their cost and performance. However, as there are several factors that our models do not consider (e.g., routing), the only way to identify the true optimal trade-offs is by running synthesis and place-and-route for each of the designs, taking into account the target platform. This is typically quite time consuming, and an exhaustive evaluation might not always be feasible in practice.

In this section, we first briefly describe the process to generate RTL from a DSL expression. Then, we evaluate the different sorting networks produced by our generator for input sizes 256 and 2048 on an FPGA.

5.1. RTL Generation from DSL Expressions

A sorting network expressed in our DSL together with input set size (n) and implementation directives (w and d) completely specify a set of sorting network and hardware implementation choices. We have created an automated hardware generator (an extension of Spiral [Milder et al. 2012; Püschel et al. 2005]) that produces annotated DSL expressions and compiles them to synthesizable RTL Verilog (which is suitable for synthesis using standard FPGA or ASIC design tools). Figure 15 illustrates our system's flow.

First, one of the five breakdown rules (SN1, SN2, SN3, SN4, or SN5) is selected and adapted to reflect the user's choices for input set size n and implementation directives d and w. An intermediate code representation is generated at this point. Next, a set of optimization and rewriting rules are applied with the goal of simplifying the expressions and improving the quality of the generated design. Then, the result is translated into synthesizable RTL Verilog. During translation the design is automatically pipelined to maximize its achievable clock frequency, and timing analysis is performed to ensure that all signals route through the system and any feedback paths with correct

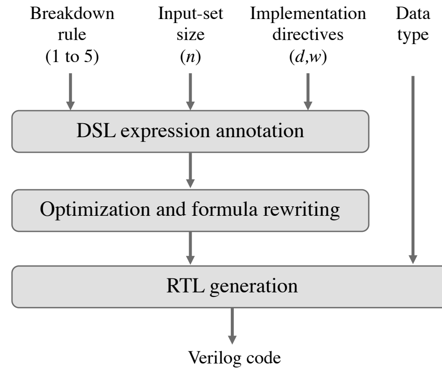


Fig. 15. Flow diagram of the generator.

timing. Other implementation options, such the data type to be sorted, are additionally fed to the RTL generation stage.

Streaming permutations (when $w < n$) are implemented following the algorithms in Püschel et al. [2009]. Permutations that operate on nonstreaming data (when $w = n$) are simply implemented with wires.

The generator additionally calculates the system's memory requirements and the latency and throughput of each block, relative to the clock frequency. It outputs the final design alongside Verilog testbenches for the verification of the created modules. An online version of the generator is available at Zuluaga et al. [2012b].

5.2. Experimental Setup

For our experiments, we generated sorting networks that process 16-bit fixed-point data. Our generator has been optimized to target the Xilinx FPGA platforms. For the experiments presented in this section, we specifically target the Xilinx Virtex-6 XC6VLX760 FPGA and use the Xilinx ISE tools (Version 13.1). This device contains 118,000 configurable slices and 720 hard on-chip memory units called Block RAM (BRAM). We characterize each design by its cost: BRAM and slice usage, and by its performance in terms of latency or throughput. Latency is measured in microseconds, and throughput is measured in giga samples per second; both are calculated based on the design's maximum execution frequency, given by Xilinx ISE after place-and-route. BRAM and FPGA slice usage are also taken from post-place-and-route reports. The synthesis and place-and-route processes were configured to maximize execution frequency.

5.3. Exploring the Design Space

Using our sorting network generator, we obtained all possible Verilog designs for $n = 256$ and $n = 2048$ by exploring the different breakdown rules and implementation directives in Table 3.3. Designs with streaming width greater than 256, and clearly suboptimal d_i configurations for SN2 and SN4 were not considered. A total of 199 designs were generated for $n = 256$, and 215 designs were generated for $n = 2048$; the generation process took a total of only a few seconds. Synthesis and place-and-route were attempted for all of the generated designs. From these, 181 designs for $n = 256$ and 156 for $n = 2048$ fit onto the FPGA. This process took from minutes to hours, depending on the complexity of the implementation. For comparison purposes, we manually implemented and evaluated a linear sorter for both $n = 256$ and $n = 2048$.

Area-throughput trade-off. The plots in Figure 16 show throughput and area, in a logarithmic scale, for the designs that fit onto the target FPGA. In each plot, the x-axis is the number of FPGA slices used, the y-axis is the throughput, and the size of the marker is proportional to the number of BRAMs used. For our analysis, the Pareto-optimal designs are selected based on throughput and

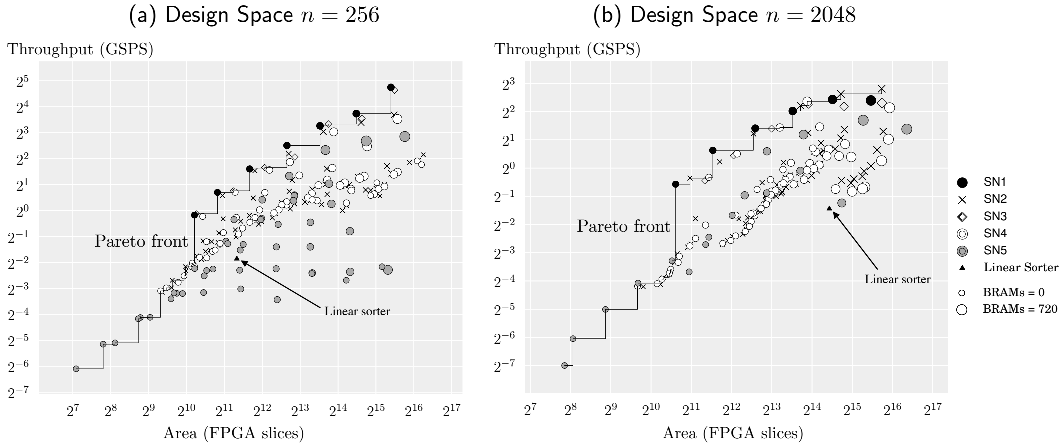


Fig. 16. Area and throughput of the designs generated for S_{256} and S_{2048}.

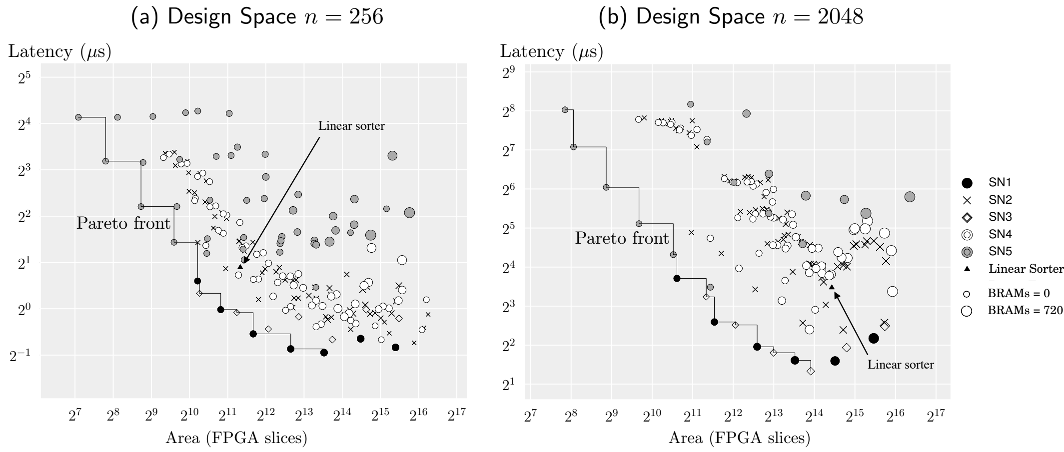


Fig. 17. Area and latency of the designs generated for S_{256} and S_{2048}.

FPGA slices used, and they are connected with a line. Only the Pareto-optimal designs need to be considered for an application if throughput and FPGA slices are the main design objectives. However, BRAM usage could be considered as a third design objective, in which case some additional points could become Pareto-optimal. These experiments confirm that SN1 produces mostly optimal fully streaming designs, SN5 produces the smallest optimal designs, and SN4 generates a few optimal designs between these two trade-offs. Breakdown rules SN2 and SN3 that were categorized as suboptimal in the quantitative analysis presented in Section 4 generate only a few optimal designs and could be omitted without losing important trade-offs.

Fully streaming designs were obtained with streaming width of up to 128 for $n = 256$ and 64 for $n = 2048$. For $n = 2048$, designs with $w = 64$ and $w = 32$ achieve approximately the same throughput, because as the complexity of the designs grow, maximum working frequencies may decrease depending on the target platform.

The linear sorter only yields one alternative and is far suboptimal for both $n = 256$ and $n = 2048$.

Area-latency trade-off. Depending on the application, the latency of the implementation may be more important than its throughput. In this case, the latency-area trade-off becomes relevant. Figure 17 shows latency and area, in a logarithmic scale, for the designs that fit onto the target

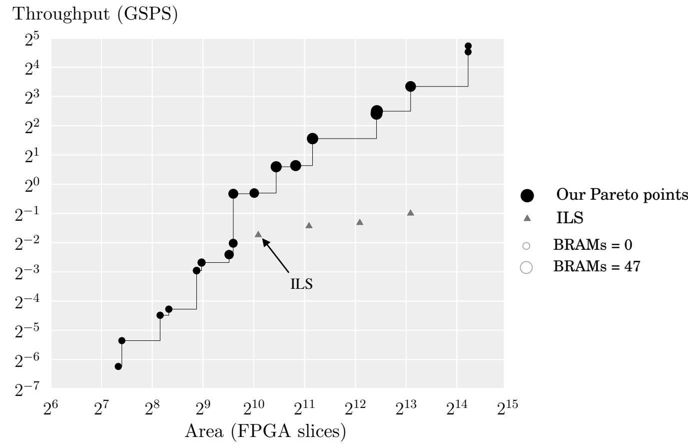


Fig. 18. Comparison of the Pareto-optimal designs obtained with our exploration and the ones obtained with the ILS proposed in Ortiz and Andrews [2010], for $n = 64$.

FPGA. In each plot, the x-axis is the number of slices used, the y-axis is the latency, and the size of the marker is proportional to the number of BRAMs used. Again a line connects the Pareto-optimal designs. In both cases, $n = 256$ and $n = 2048$, the smallest optimal designs, generated with SN5, have the highest latency. On the other hand, the optimal designs with lowest latencies are obtained with SN1 and SN3, which generate fully streaming designs that require more resources. No optimal designs were found with either SN2 or SN4. The plots in Figure 17 also include the linear sorter, which again is far suboptimal.

These experiments show that our designs offer a wide range of optimal trade-offs that are evenly spread over a wide range of cost and performance, from small low-throughput designs to large high-throughput designs. This gives designers the flexibility to choose the implementation that best suits the goals and constraints of the entire system.

5.4. Comparison to Other Work

Next, we compare our designs with the interleaved linear sorters (ILS) presented in Ortiz and Andrews [2010]. For $n = 64$, Figure 18 shows throughput and area of our Pareto-optimal designs connected by a line and the ILS designs with streaming widths $w = 1, 2, 4, 8$ using the results shown in Ortiz and Andrews [2010]. To allow for a fair comparison, we synthesized our designs targeting the same FPGA platform that was used in their experiments. This experiment shows that increasing streaming width in the ILS (as an attempt to trade area for throughput) only causes a very slight increase in throughput. Overall, our designs provide higher throughput with lower cost and are highly scalable, allowing a wide range of Pareto-optimal trade-offs.

6. CONCLUSIONS

We presented bitonic streaming sorting networks, a new class of hardware structures for sorting with novel and improved cost-performance trade-offs compared to prior work. Our goal of this work was completeness—that is, we wanted to present an analysis that satisfies the theoretician and the application engineer. At the theoretical level, we provided the formalism to describe our networks and performed an asymptotic and exact analysis of their cost and performance. At the application level, we confirmed and refined the analysis with real hardware implementations of our networks on FPGAs. The link between theory and application was established with our domain-specific hardware generator, which allowed us to obtain and analyze thousands of implementations with little effort.

The first practical key contribution is in the wide range of cost-performance trade-offs that our networks offer. These improve over prior work and allow the engineer to choose the design that best matches the application constraints. The second and possibly most salient contribution of our work is the ability to shrink the area cost of the logic to sort to any $O(w)$ including $O(1)$, moving the costly $O(n)$ component completely to RAM cells, which scale significantly better in real-world terms. We achieved this while still keeping efficient throughput and latency. This means that with our work hardware sorting becomes feasible for much larger data sets as we demonstrated with our experiments.

APPENDIX

A. STREAMING PERMUTATIONS

This section provides brief background on the streaming permutation techniques developed by Püschel et al. [2009] and then applies them to the permutations needed in our streaming sorting networks.

We write a permutation on $n = 2^t$ data elements as a $2^t \times 2^t$ matrix, e.g., U_{2^t}, in which each row and column contains exactly one value of one and all the remaining entries are zero. The permuted vector y is obtained by multiplying this matrix with a given input vector x (i.e. $y = U_{2^t}x$).

One simple example from our DSL (Section 2) is the stride-by-two permutation on four elements:

$$L_4^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which maps elements 0 and 3 to themselves and exchanges 1 and 2.

A different point of view does not consider the indices $0, 1, 2, 3$ but their binary representation. To do so, we represent each index by a column vector of t elements in the field $\mathbb{F}_2 = \{0, 1\}$. The first element of the vector is the most significant bit; for example 2 is written as $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$.

We denote with $\boldsymbol{x}_i$ the binary representation of i, $0 \le i < 2^t$, and with $\boldsymbol{y}_i$ the binary representation of the corresponding output index. To continue our simple example, for L_4^2, we have the following input indices,

$$\boldsymbol{x}_0 = \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right], \quad \boldsymbol{x}_1 = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right], \quad \boldsymbol{x}_2 = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right], \quad \boldsymbol{x}_3 = \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right],$$

and output indices,

$$\boldsymbol{y}_0 = \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right], \quad \boldsymbol{y}_1 = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right], \quad \boldsymbol{y}_2 = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right], \quad \boldsymbol{y}_3 = \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right].$$

We say that a permutation U_{2^t} is *linear* in the binary representation of the data locations (or simply linear) if there is a matrix $\boldsymbol{U}_t$ such that $\boldsymbol{y}_i = \boldsymbol{U}_t\boldsymbol{x}_i$, for all $0 \le i < 2^t$. Necessarily, $\boldsymbol{U}_t$ is invertible. Addition and multiplication in $\mathbb{F}_2$ are the "xor" and "and" operation, respectively. Note that the linearity property implies that 0 is mapped to itself; thus, most permutations are not linear.[1]

We use the function π to represent the mapping from an invertible bit matrix $\boldsymbol{U}_t$ to the corresponding permutation matrix U_{2^t}. To distinguish a permutation matrix from its corresponding bit transformation matrix, we boldface the latter. Thus, $\pi : \boldsymbol{U}_t \mapsto U_{2^t}$ or $\boldsymbol{U}_t = \pi^{-1}(U_{2^t})$.

To continue our previous example of the stride permutation, we can see that

$$\pi^{-1}(L_4^2) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

We can easily verify this equation by multiplying $\pi^{-1}(L_4^2)$ with each $\boldsymbol{x}_i$ and see that it produces the correct value for $\boldsymbol{y}_i$.

The following lemma from Püschel et al. [2009] states some of the known properties of π.

[1] To be more precise, there are as many linear permutations on $n = 2^t$ points as there are invertible $t \times t$ matrices over $\mathbb{F}_2$, namely $\prod_{0 \le i < t}(2^t - 2^i)$.

(a) $I_{2^{t-1}} \oplus J_{2^{t-1}}$

$$\begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ & & & & & & & 1 \\ & & & & & & 1 & \\ & & & & & \iddots & & \\ & & & & 1 & & & \end{bmatrix}$$

(b) $\boldsymbol{T}_t = \pi^{-1}(I_{2^{t-1}} \oplus J_{2^{t-1}})$

$$\begin{bmatrix} 1 & & & \\ 1 & 1 & & \\ \vdots & & \ddots & \\ 1 & & & 1 \end{bmatrix}$$

Fig. 19. (a) Permutation matrix $I_{2^{t-1}} \oplus J_{2^{t-1}}$. (b) Associated bit transformation matrix $\boldsymbol{T}_t = \pi^{-1}(I_{2^{t-1}} \oplus J_{2^{t-1}})$.

LEMMA A.1. *Let S, N be permutation matrices that are linear. Then, the following holds.*

(1) $\pi^{-1}(SN) = \pi^{-1}(S)\pi^{-1}(N)$
(2) $\pi^{-1}(S \otimes N) = \pi^{-1}(S) \oplus \pi^{-1}(N)$
(3) $\pi^{-1}(I_{2^t}) = \boldsymbol{I}_t$
(4) $\pi^{-1}(L_{2^t}^{2^k}) = \boldsymbol{C}_t^{t-k}$

where C_t^{t-k} is a cyclic shift by $t-k$ on t points or the $(t-k)$th power of the cyclic shift by 1 C_t. C_t is a permutation that maps position i to position $i + 1 \bmod t$.

An important permutation occurring in sorting networks is $I_{2^{t-1}} \oplus J_{2^{t-1}}$ (illustrated in Figure 19(a); we omit zero entries). This permutation is linear; the corresponding bit transformation matrix $\boldsymbol{T}_t$ is shown in Figure 19(b):

LEMMA A.2. $\pi^{-1}(I_{2^{t-1}} \oplus J_{2^{t-1}}) = \boldsymbol{T}_t$.

PROOF. The permutation $I_{2^{t-1}} \oplus J_{2^{t-1}}$ maps the top half of the input vector to the same location, and inverts the location of the second half. The first half, $i < n/2$, are those indices whose bit representation has first bit zero. For those to stay unchanged, the last $t-1$ columns of $\boldsymbol{T}_t$ have to be an identity as shown in Figure 19(b).

The second half, $i \geq n/2$, are those indices whose bit representation has first bit one. This bit has to stay unchanged, and the rest of the bits have to be inverted. From this we derive that the first column of $\boldsymbol{T}_t$ should be all ones. □

The permutations that are used in SN1–5 are: $P_{2^t}^{2^j}$, $R_{2^t}^{2^i} Q_{2^t}^{2^i}$, $R_{2^t}^{2^i}$, and $L_{2^t}^{2^{t-1}}$. Using Lemmas A.2 and Lemmas A.1 we obtain that all these permutations are linear. Therefore, all five sorting networks considered in this article can be streamed with any streaming width $w \mid 2^t$, $w \geq 2$, using the techniques in Püschel et al. [2009]. Table VI shows the bit transformation matrices corresponding to each of the permutation used in SN1–5. Additionally, the table specifies the sorting networks that use them and whether the bit transformation matrix is itself a permutation matrix. This information is relevant in the following.

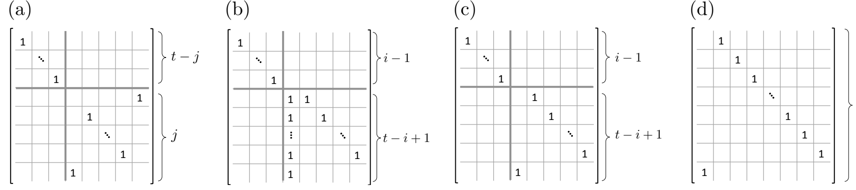
B. COST OF STREAMING PERMUTATIONS

In Table IV in Section 3, we provided the number of switches and RAM words required in the streaming permutation stages of SN1 through 5, which are used in our analysis. In this section, we explain how these values are derived for the occurring permutations.

First, it is necessary to understand permutations of the form $U_n = I_m \otimes U_s$. This expression indicates that U_s (a permutation on s elements) is performed m times in parallel over the $ms = n$ data elements. Examples of these permutations include $P_{2^t}^{2^j}$, $R_{2^t}^{2^i} Q_{2^t}^{2^i}$, $R_{2^t}^{2^i}$. When streaming reuse is applied on a permutation U_n of this form, the cost of its implementation may be decreased depending on how w and U_s split the input set. This is formalized in the following lemmas.

Table VI. Permutations That Occur in SN1 Through SN5 and Their Associated Bit Transformation Matrices

Permutation	Used in SN	Associated Bit Matrix		
		Structured Expression	Visualized	Is Permutation on Bits
$P_{2^t}^{2^j}$	1,3	$I_{t-j} \oplus ((1 \oplus C_{j-1}^1) C_j^{j-1})$	(a)	yes
$R_{2^t}^{2^i} Q_{2^t}^{2^i}$	1,2	$I_{i-1} \oplus (C_{t-i+1}^1 T_{t-i+1})$	(b)	no
$R_{2^t}^{2^i}$	2,3,4	$I_{i-1} \oplus (C_{t-i+1}^1)$	(c)	yes
$L_{2^t}^{2^{t-1}}$	5	C_t^1	(d)	yes



LEMMA B.1. *Let $U_n = I_m \otimes U_s$ be a permutation to be streamed with streaming width w, where s evenly divides w. Then the number of switches and the number of memory words required to implement U_n is zero.*

PROOF. This is the standard case of streaming reuse represented in Fig 9. Since $w \geq s$, the permutation U_s does not have to be streamed because all s elements are available at the same time. Thus U_s is implemented simply with wires w/s times in parallel to support a total of w words per clock cycle. □

LEMMA B.2. *Let $U_n = I_m \otimes U_s$ be a permutation to be streamed with streaming width $w < s$, where w evenly divides s. Then, the cost of implementing U_n with streaming width w is equal to the cost of implementing U_s with the same streaming width w.*

Lemma B.2 is demonstrated in Püschel et al. [2009] (see Section 6, Example 3, Case 2).

Next, we sketch how to derive the number of memory words and the number of switches required for the implementation of a streaming permutation U on $n = 2^t$ words with streaming width $w = 2^k$ in the architecture shown in Figure 10(b).

Number of RAM words. Every memory bank must store $2n/w$ words, and thus a total storage of $2n$ words is required.

Number of switches. The connection networks that are placed before and after the blocks of RAM (see Figure 10(b)) consist of basic switches. The networks determine how the data is stored in the RAMs and how the data is read from the RAMs. To explain how the networks are obtained for a given permutation U, we first partition the associated bit matrix $\boldsymbol{U}$ according to the desired streaming width $w = 2^k$ as

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{U}_4 & \boldsymbol{U}_3 \\ \boldsymbol{U}_2 & \boldsymbol{U}_1 \end{bmatrix}$$

where $\boldsymbol{U}_1$ is a $k \times k$ matrix.

To obtain the two networks (for the write and read stage), $\boldsymbol{U}$ has to be factored as $\boldsymbol{U} = \boldsymbol{N}^{-1}\boldsymbol{M}$, such that both sub-blocks (defined as for $\boldsymbol{U}$ above) $\boldsymbol{M}_1$ and $\boldsymbol{N}_1$ are invertible. The cost of the two networks is then determined by rank($\boldsymbol{M}_2$) and rank($\boldsymbol{N}_2$). Specifically, the write stage requires 2^{k-1}rank($\boldsymbol{M}_2$) switches and the read stage requires 2^{k-1}rank($\boldsymbol{N}_2$) switches. Püschel et al. [2009] present an algorithm to find $\boldsymbol{N}$ and $\boldsymbol{M}$ while minimizing the rank of $\boldsymbol{N}_2$ and $\boldsymbol{M}_2$.

For permutations where $\boldsymbol{U}$ is itself a permutation matrix (such as $P_{2^t}^{2^j}$, $R_{2^t}^{2^i}$ and $L_{2^t}^{2^{t-1}}$), the number of switches obtained with the algorithm is optimal. In this case, the number of switches required in

the read stage is $2^{k-1}(k - \text{rank}(\boldsymbol{U}_1))$ and the number of switches required in the write stage is $2^{k-1}(k - \text{rank}(\boldsymbol{U}_1'))$, where $\boldsymbol{U}_1'$ is the lower-right block of $\boldsymbol{U}^{-1}$. As the matrices are partitioned according to k, the ranks of their submatrices, and thus the cost of streaming the permutation, vary with the streaming width. In summary, we use the following steps to find the number of switches to stream a permutation U.

(1) If $U = I \otimes U_s$, find $\boldsymbol{U} = \pi^{-1}(U_s)$. Otherwise find $\boldsymbol{U} = \pi^{-1}(U)$.
(2) If $\boldsymbol{U}$ is a permutation matrix, calculate the rank of $\boldsymbol{U}_1$ and the rank of $\boldsymbol{U}_1'$. The number of switches is then $2^{k-1}(k - \text{rank}(\boldsymbol{U}_1))$ for the write stage and $2^{k-1}(k - \text{rank}(\boldsymbol{U}_1'))$ for the read stage.
(3) Otherwise, find a suitable factorization $\boldsymbol{U} = \boldsymbol{N}^{-1}\boldsymbol{M}$, as explained in Püschel et al. [2009], and calculate the rank of $\boldsymbol{M}_2$ and $\boldsymbol{N}_2$. The number of switches is then $2^{k-1}\text{rank}(M_2)$ for the write stage and $2^{k-1}\text{rank}(N_2)$ for the read stage.

Using the preceding steps, we obtain the number of switches and the number of memory words required for each permutation occurring in our sorting networks. Through summation over the iteration variables i and j we then obtain the total numbers shown in the last two columns in Table IV for each sorting network.

REFERENCES

M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An O(N Log N) Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. ACM, New York, NY, USA, 1–9. DOI: http://dx.doi.org/10.1145/800061.808726

K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS '68 (Spring))*. ACM, New York, NY, USA, 307–314. DOI: http://dx.doi.org/10.1145/1468075.1468121

Gianfranco Bilardi and Franco P Preparata. 1984. An Architecture for Bitonic Sorting with Optimal VLSI Performance. *IEEE Trans. Comput.* 100, 7 (1984), 646–651. DOI: http://dx.doi.org/10.1109/TC.1984.5009338

Ren Chen, Sruja Siriyal, and Viktor Prasanna. 2015. Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 240–249. DOI: http://dx.doi.org/10.1145/2684746.2689068

Yen-Cheng Chen and Wen-Tsuen Chen. 1994. Constant Time Sorting on Reconfigurable Meshes. *IEEE Trans. Comput.* 43, 6 (June 1994), 749–751. DOI: http://dx.doi.org/10.1109/12.286307

Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. 1989. The Periodic Balanced Sorting Network. *J. ACM* 36, 4 (Oct. 1989), 738–757. DOI: http://dx.doi.org/10.1145/76359.76362

Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC) (Lecture Notes in Computer Science)*, Vol. 5658. Springer-Verlag, Berlin, Heidelberg, 385–410. DOI: http://dx.doi.org/10.1007/978-3-642-03034-5_18

Ju-Wook Jang and Viktor K. Prasanna. 1992. An Optimal Sorting Algorithm on Reconfigurable Mesh. In *Parallel Processing Symposium*. IEEE, Beverly Hills, CA, 130–137. DOI: http://dx.doi.org/10.1109/IPPS.1992.223059

J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. 1990. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. *Circuits, Systems, and Signal Processing* 9, 4 (1990), 449–500. DOI: http://dx.doi.org/10.1007/BF01189337

Donald E. Knuth. 1968. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley Pub. Co.

Miroslaw Kutylowsky, Krzysztof Loryś, Brigitte Oesterdiekhoff, and Rolf Wanka. 2000. Periodification scheme: Constructing sorting networks with constant period. *J. ACM* 47, 5 (Sept. 2000), 944–967. DOI: http://dx.doi.org/10.1145/355483.355490

Christophe Layer and Hans-Jörg Pfleiderer. 2004. A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data. In *Field Programmable Logic and Application*. Lecture Notes in Computer Science, Vol. 3203. Springer, Berlin, Heidelberg, 648–657. DOI: http://dx.doi.org/10.1007/978-3-540-30117-2_66

Christophe Layer, Daniel Schaupp, and Hans-Jörg Pfleiderer. 2007. Area and Throughput Aware Comparator Networks Optimization for Parallel Data Processing on FPGA. In *International Symposium on Circuits and Systems*. IEEE, New Orleans, LA, 405–408. DOI: http://dx.doi.org/10.1109/ISCAS.2007.378475

Chen-Yi Lee and Jer-Min Tsai. 1995. A Shift Register Architecture for High-Speed Data Sorting. *Journal of VLSI Signal Processing Systems* 11, 3 (Dec. 1995), 273–280. DOI: http://dx.doi.org/10.1007/BF02107058

F.T. Leighton. 1992. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Vol. 1. M. Kaufmann Publishers, San Mateo, CA.

Chi-Sheng Lin and Bin-Da Liu. 2002. Design of a Pipelined and Expandable Sorting Architecture with Simple Control Scheme. In *International Symposium on Circuits and Systems*. IEEE, IV–217–IV–220. DOI:http://dx.doi.org/10.1109/ISCAS.2002.1010428

Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems* 17, 2, Article 15 (2012), 33 pages. DOI:http://dx.doi.org/10.1145/2159542.2159547

Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting Networks on FPGAs. *The VLDB Journal* 21, 1 (Feb. 2012), 1–23. DOI:http://dx.doi.org/10.1007/s00778-011-0232-z

Jorge Ortiz and David Andrews. 2010. A Configurable High-throughput Linear Sorter System. In *Reconfigurable Architectures Workshop at International Symposium on Parallel and Distributed Systems*. IEEE, Atlanta, GA. DOI:http://dx.doi.org/10.1109/IPDPSW.2010.5470730

Marshall C. Pease. 1968. An Adaptation of the Fast Fourier Transform for Parallel Processing. *J. ACM* 15, 2 (April 1968), 252–264. DOI:http://dx.doi.org/10.1145/321450.321457

Roberto Perez-Andrade, Rene Cumplido, Claudia Feregrino-Uribe, and Fernando Martin Del Campo. 2009. A Versatile Linear Insertion Sorter Based on an FIFO scheme. *Microelectronics Journal* 40, 12 (Dec. 2009), 1705–1713. DOI:http://dx.doi.org/10.1016/j.mejo.2009.08.006

Markus Püschel, Peter A. Milder, and James C. Hoe. 2009. Permuting Streaming Data Using RAMs. *J. ACM* 56, 2, Article 10 (April 2009), 34 pages. DOI:http://dx.doi.org/10.1145/1502793.1502799

Markus Püschel, Peter A. Milder, and James C. Hoe. 2012. System and method for designing architecture for specified permutation and datapath circuits for permutation. (Nov. 2012). US Patent No. 8,321,823, Filed Oct. 2nd., 2008, Issued Nov. 27th., 2012.

Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232–275. DOI:http://dx.doi.org/10.1109/JPROC.2004.840306

Isaac D. Scherson and Sandeep Sen. 1989. Parallel Sorting in Two-Dimensional VLSI Models of Computation. *IEEE Trans. Comput.* 38, 2 (February 1989), 238–249. DOI:http://dx.doi.org/10.1109/12.16500

H.S. Stone. 1971. Parallel Processing with the Perfect Shuffle. *IEEE Trans. Comput.* 20, 2 (1971), 153–161. DOI:http://dx.doi.org/10.1109/T-C.1971.223205

Charles Van Loan. 1992. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA.

Yanjun Zhang and S. Q. Zheng. 2000. An Efficient Parallel VLSI Sorting Architecture. *VLSI Design* 11, 2 (2000), 137–147. DOI:http://dx.doi.org/10.1155/2000/14617

Marcela Zuluaga, Peter Milder, and Markus Püschel. 2012a. Computer Generation of Streaming Sorting Networks. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1245–1253. DOI:http://dx.doi.org/10.1145/2228360.2228588

Marcela Zuluaga, Peter Milder, and Markus Püschel. 2012b. Sorting Network IP Generator. (2012). http://www.spiral.net/hardware/sort/sort.html

Received October 2015; accepted November 2015