Computer Generation of Streaming Sorting Networks

Marcela Zuluaga
ETH Zurich
zuluaga@inf.ethz.ch

Peter Milder
Carnegie Mellon University
pam@ece.cmu.edu

Markus Püschel
ETH Zurich
pueschel@inf.ethz.ch

ABSTRACT

Sorting networks offer great performance but become prohibitively expensive for large data sets. We present a domain-specific language and compiler to automatically generate hardware implementations of sorting networks with reduced area and optimized for latency or throughput. Our results show that the generator produces a wide range of Pareto-optimal solutions that both compete with and outperform prior sorting hardware.

Categories and Subject Descriptors

B.5.2 [**Register Transfer Level Implementation**]: Design Aids—*automatic synthesis*; F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms

Design

Keywords

Hardware Sorting, Design Space Exploration, HDL Generation

1. INTRODUCTION

Sorting is a fundamental operation that is required in a wide range of applications with different requirements in performance and cost. For example, continuous data processing applications may have throughput requirements that cannot be matched by sequential sorting algorithms running on a processor. On the other hand, applications with more relaxed performance requirements can benefit from offloading of time consuming operations such as sorting, by running them on dedicated hardware that can be implemented using simple components.

Two main categories of hardware sorters can be distinguished: sorting networks and linear sorters. Sorting networks operate in parallel over the input elements, processing them through a network of comparison-exchange elements. This solution offers great performance but becomes prohibitively expensive for large data sets because of the area cost or since elements can no longer be provided at the same time. Instead, linear sorters input one element at a time, sorting them into a register array as they are received. This approach can offer small designs but fails to scale in performance.

In this paper, we fill the gap between these two approaches by introducing sorting networks that can offer a variable streaming width, while maintaining high throughput capabilities. We show that this approach dramatically reduces the footprint of sorting networks by effectively reusing hardware components to operate sequentially over the input elements. To the best of our knowledge, this type of reuse had not been done for sorting networks. The likely reason is in the challenge of streaming the permutation stages. We solve this problem using the method in [14].

Furthermore, we built a hardware generator that enables the automatic and systematic exploration of a large design space for hardware sorters. This way, designers can identify Pareto-optimal solutions that match their specific requirements. The generator makes use of a small Domain-Specific Language (DSL) to represent at a high level different known sorting networks. These DSL expressions are then annotated with implementation directives that capture different types and degree of reuse, which gives rise to a large and complex design space. Finally, a special DSL compiler generates Register-Transfer Level (RTL) descriptions for each desired design.

In summary, the contributions of this paper are

- to our knowledge the first high throughput streaming sorting networks with arbitrary streaming width;
- a DSL-based framework to describe sorting networks and architectural parameters;
- a hardware description generator that automatically translates DSL expressions into a hardware description language, thus enabling the automatic exploration of a large design space to identify Pareto-optimal solutions; and
- a wide range of hardware solutions for sorting that are highly optimized for area, throughput, latency and memory usage.

1.1 Related Work

Many sorting methods have been invented and studied in detail in the past decades [5, 8]. Among those, sorting networks are attractive due to their inherent parallelism and input-independent structure. Bitonic sorting networks, first presented in [2], have a regular structure and can sort $n = 2^t$ elements in $\frac{1}{2}\log_2 n(\log_2 n + 1)$ stages of $\frac{n}{2}$ parallel compare operations. At every stage, elements are permuted such that a sorted sequence is obtained at the final stage. Although other approaches have shown to require $O(\log_2 n)$ stages [1], the lack of regularity and the large constants hidden in this bound make them in practice less efficient than bitonic networks [12].

Several techniques have been proposed to reduce the area of sort-

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

ing networks. In [15], it is demonstrated that a single physical stage, composed of an array of $\frac{n}{2}$ compare modules followed by a perfect shuffle permutation, can be used to sort n elements by recirculating them $(\log_2 n)^2$ times. A version of this implementation that can support several input sizes (up to some maximum) is presented in [6] but does not gain performance. Reconfigurable logic is used to change the required shuffles depending on the input width. The work in [7] explores area-throughput trade-offs with a pipelined architecture composed of $\log_2 n$ stages that recirculates the data within each stage. Our generator can produce each of these designs as special case. Further, we will show that through our novel approach to implementing reuse, we can considerably further improve the area/performance tradeoff and at the same time offer a much larger space of Pareto-optimal designs.

Not based on sorting networks is [11], which uses linear sorters [8] with the goal to improve throughput. The authors show how a stack of linear sorters can be used to parallelize the sorting of a joint list by interleaving the inputs of the system and the outputs of the individual linear sorters. Results show that interleaving creates significant delays due to conflicts and increases the complexity of the design, such that input widths greater than eight no longer represent a performance gain due to the low execution frequency of the resulting datapath. We include this work in our comparison and again show considerable and systematic gains in area and performance.

Finally, as one application example of hardware sorters, we cite [10], which evaluates various sorting networks on FPGAs, focusing on accelerating database applications. The authors demonstrate that sorting networks on FPGAs represent a competitive solution to sorting data sets of small sizes; our work may help to remove this restriction.

2. DSL FOR SORTING NETWORKS

In this section we introduce a small domain-specific language (DSL) to represent sorting networks, borrowing concepts from [3] and [9]. The elements of the language are structured operators (viewed as data flow graphs) that map vectors to vectors of the same length. For instance, the operator S_2 transforms a vector of two elements into a vector of two ordered elements. Each operator is either a first order operator or a composition using higher order operators. In the following, vectors are written as $x = (x_i)_{0\le i<n} = (x_0, \ldots, x_n)$, and operators A_n are functions on these vectors of length n. Thus, $y = A_n x$ indicates that an operator A_n is applied to an input vector x, generating an output vector y. We formally introduce first order operators and higher order operators next.

First order operators. We define the following first order operators:

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

I_n is the identity operator, J_n flips the input vector, and L_m^n performs a stride-by-m permutation on n elements (also called corner turn, transposition, or shuffle). $L_{n/2}^n$ is called perfect shuffle. S_2 and X_2^c are the basic kernels for sorting networks. S_2 sorts 2 elements into ascending order, whereas X_2^c can be configured to perform ascending sorting, descending sorting, or to preserve the original order of the input elements.

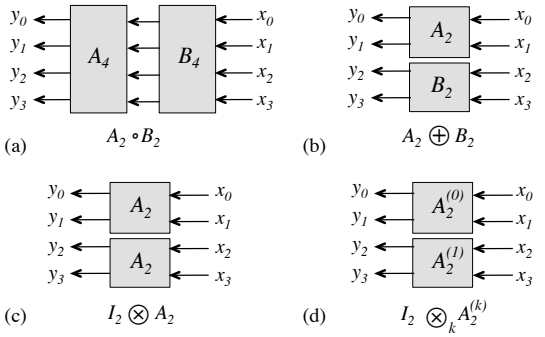

Figure 1: Higher order operators and associated data-flow graph structures: (a) composition, (b) direct sum, (c) tensor product, and (d) parameterized tensor product.

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

- Composition ($\circ$): $A_n \circ B_n$ is the composition of operators, as shown in Fig. 1(a): the input vector is first mapped by B and then by A. The symbol $\circ$ may be omitted to simplify expressions. For an iterative composition we use the product sign: $A_n^{(0)} \circ \cdots \circ A_n^{(t-1)} = \prod_{i=1}^{t} A_n^{(i)}$. Since composition is applied from right to left, we draw dataflow diagrams accordingly.
- Direct sum ($\oplus$): $A_n \oplus B_m$ signifies that A_n maps the upper n elements and B_m the bottom m elements of the input vector (see Fig. 1(b)).
- Tensor product ($\otimes$): The expression $I_m \otimes A_n = A_n \oplus \cdots \oplus A_n$ replicates m times the operator A_n to operate in parallel on the input vector (see Fig. 1(c)).
- Indexed tensor product ($\otimes_k$): The expression $I_m \otimes_k A_n^{(k)} = A_n^{(0)} \oplus \cdots \oplus A_n^{(m-1)}$ allows for a change in the replicated operator through the parameter k (see Fig. 1(d)).

Bitonic sorting networks: Example. Our DSL is designed to represent sorting networks with a regular structure, specifically those based on bitonic sorting [5, pp. 230]. We define S_n as the sorting operator that transforms an input vector of size n into a sorted ascending sequence of the same size. We assume $n = 2^t$ is a two-power. Further, we define M_n as a bitonic merge operator that transforms a regular-bitonic sequence of size n into a sorted sequence of the same size. A regular-bitonic sequence is a concatenation of an ascending sequence and a descending sequence of size $n/2$. In the case of $n = 2$, $M_2 = S_2$.

The classical bitonic sorting network [2] consists of a sequence of t merging stages. In our DSL it becomes

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

Specifically for $n = 8$,

$$S_8 = M_8(I_2 \otimes J_4)(I_2 \otimes M_4)(I_2 \otimes (I_2 \oplus J_2))(I_4 \otimes M_2), \quad (2)$$

and the associated dataflow graph is shown in Fig. 2(a).

Each merger in (1) is recursively decomposed into smaller mergers. In our DSL,

$$M_{2^t} = (I_2 \otimes M_{2^{t-1}}) L_2^{2^t} (I_{2^{t-1}}^2 \otimes S_2) L_{2^{t-1}}^{2^t}. \quad (3)$$

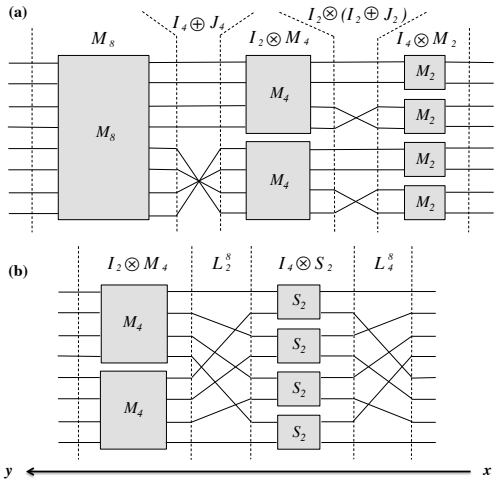

Figure 2: A bitonic sorter for $n = 8$: (a) S_8 is based on (1) and sorts by a sequence of mergers; each merger is again decomposed. For example, (b) M_8 is based on (3).

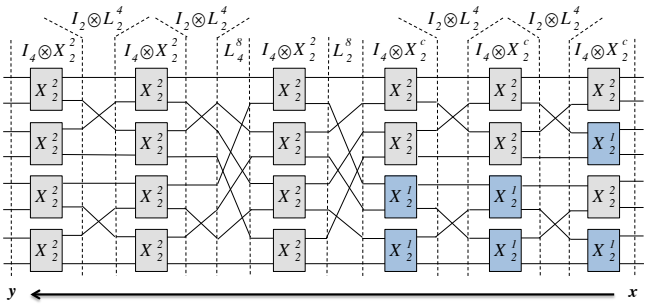

Figure 3: Data flow graph representation of a bitonic sorter of size 8 based on SN3. X_2^1 sorts the inputs in descending order and X_2^2 sorts the inputs in ascending order.

Fig. 2(b) shows the example M_8.

Recursive expansion of (3) yields a complete decomposition into basic blocks S_2:

$$M_{2^t} = \prod_{j=1}^{t} \left[(I_{2^{t-j}} \otimes L_2^{2^j})(I_{2^{t-1}} \otimes S_2)(I_{2^{t-j}} \otimes L_{2^{j-1}}^{2^j}) \right]. \quad (4)$$

Note that each of the t stages has a different permutation that connects the parallel S_2's. Similar to the Pease FFT [13], this expression can be manipulated using tensor product identities [4] into the "constant geometry" form

$$M_{2^t} = \prod_{j=1}^{t} \left[(I_{2^{t-1}} \otimes S_2) L_{2^{t-1}}^{2^t} \right]. \quad (5)$$

The permutation is now the same in each iteration. This manipulation is also applied to sorting networks in [6].

A complete sorting network is obtained by inserting either (4) or (5) into (1). Besides these two, several other variants with slightly different structure and number of stages have been reported in the literature. In our generator, we consider the following five sorting networks; the corresponding DSL expressions are in Table 1.

- SN1 is obtained by inserting (4) into (1) [2].
- SN2 is obtained by inserting (5) into (1) [6]. By using (5) as a breakdown rule for M_n, the regularity of the design increases, but it also increases the complexity of the occurring permutations.
- SN3 is obtained by rewriting SN1 to use the operator X_2^c instead of S_2 to eliminate the J_n permutations [2]. SN3 represents the trade-off of eliminating a permutation at the cost of the additional control logic for X_2^c. SN3 for $n = 8$ is shown in Fig. 3.
- SN4 is obtained by rewriting SN2 to use the operator X_2^c, eliminating the permutations J_n [6].
- Each of the prior networks has $\frac{t}{2}(t+1)$ stages with different permutations in each stage. SN5 has t^2 stages but perfect regularity [15]. Each stage consists of parallel pairwise comparisons, followed by the same perfect shuffle. Thus, SN5 increases the latency of S_n in exchange of the regularity in the permutation stages.

3. HARDWARE GENERATOR

The DSL expressions in Table 1 represent algorithms that can directly be translated into combinatorial logic. With proper pipelining, these designs offer maximal performance, but the area cost of these implementations quickly becomes prohibitive as n increases. To solve this problem our generator considers a much richer design space that arises from exploiting the regularity of the sorting networks to "fold" them by reusing sorting elements S_2 and X_2^c. We explain the procedure and the overall generator in the following.

Datapath reuse. The sorting networks in Table 1 exhibit regularity expressed through the iterative composition ($\prod$) and the tensor product ($\otimes$). These types of regularity can be mapped to a variety of sequential datapaths [9] as explained next.

First, $\prod$ indicates that an operator will be used in sequence more than once. Fig. 4 (top-left) illustrates a direct interpretation: each iteration gives an independent module in series. However, by employing *iterative reuse*, this same computation can be performed by building a single A_n module and recirculating the data around it as many times as required, as shown in Fig. 4 (top-right). In our DSL, we express this freedom by annotating the formula with a *depth* parameter d that indicates the number of modules to be implemented. A depth d smaller than the maximum t in $\prod_{j=1}^{t} A_n$ is possible if d evenly divides t and the permutations in the block A_n are not dependent on the iteration index j. This is the case in the inner composition in SN2 and SN4 and in both compositions in SN5.

Second, $I_k \otimes A_m$ indicates that operator A_m will be used k times in parallel. This is illustrated in Fig. 4 (bottom-left) for $k = \frac{n}{2}$, $m = 2$, where n data words flow into the system at the same time. Alternatively, we could perform the same computation by instead *folding* our data stream and datapath *vertically* as shown in Fig. 4 (bottom-right). Now the data words stream in and out of the system at a rate of 2 words per cycle, and flow through a single instance of A_2. We refer to this type of reuse as *streaming reuse*. We express it in our DSL by annotating expressions as shown in Fig. 4 with their *width* w, which indicates the number of inputs taken in each cycle. Our generator also supports $w = 1$, which requires that even the basic modules S_2 and X_2 are folded. In summary, any $w \geq 1$ that evenly divides n is a legal width for the expression $I_{n/2} \otimes A_2$.

By setting values of d and w, we specify a particular hardware implementation of an expression from our DSL, each with its own area and throughput. Larger values of d and w correspond to higher costs and throughput. Streaming and iterative reuse can also be combined. Table 2 shows for each network the number of stages in the datapath, the legal choices of d and w, and the associated number of sorters used.

SN1: $$\prod_{i=1}^{t-1}\left[(I_{2^{t-1}}\otimes S_2)\prod_{j=2}^{t-i+1}\left[\left(I_{2^{t-j}}\otimes(I_2\otimes L_{2^{j-1}}^{2^{j-2}})L_2^{2^j}\right)(I_{2^{t-1}}\otimes S_2)\right]\left(I_{2^{i-1}}\otimes\left(L_{2^{t-i+1}}^{2^{t-i}}(L_{2^{t-i}}\otimes J_{2^{t-i}})\right)\right)\right](I_{2^{t-1}}\otimes S_2)$$

SN2: $$\prod_{i=1}^{t-1}\left[\prod_{j=2}^{t-i+1}\left[(I_{2^{t-1}}\otimes S_2)(I_{2^{i-1}}\otimes L_{2^{t-i}}^{2^{t-i+1}})\right](I_{2^{i-1}}\otimes(I_{2^{t-i}}\otimes J_{2^{t-i}}))\right](I_{2^{t-1}}\otimes S_2)$$

SN3: $$\prod_{i=1}^{t-1}\left[(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})\prod_{j=2}^{t-i+1}\left[\left(I_{2^{t-j}}\otimes(I_2\otimes L_{2^{j-1}}^{2^{j-2}})L_2^{2^j}\right)(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})\right](I_{2^{i-1}}\otimes L_{2^{t-i+1}}^{2^{t-i}})\right](I_{2^{t-1}}\otimes_m X_2^{g(i,m)})$$

SN4: $$\prod_{i=1}^{t-1}\left[\prod_{j=2}^{t-i+1}\left[(I_{2^{t-1}}\otimes_m X_2^{g(i,m)})(I_{2^{i-1}}\otimes L_{2^{t-i+1}}^{2^{t-i}})\right]\right](I_{2^{t-1}}\otimes_m X_2^{g(t,m)});\quad g(i,m)=\begin{cases}1, & m[t-i]=1 \text{ and } i\neq 1\\ 2, & (m[t-i]=0 \text{ or } i=1\end{cases}$$

SN5: $$\prod_{i=0}^{t-1}\prod_{j=0}^{t-1}\left[(I_{2^{t-1}}\otimes_m X_2^{f(i,j,m)})L_{2^t}^{2^{t-1}}\right];\quad f(i,j,m)=\begin{cases}0, & t-1<j+i\\ 1, & m[t-1-j-i]=1 \text{ and } i\neq 0\\ 2, & m[t-1-j-i]=0 \text{ or } i=0\end{cases}$$

Table 1: DSL representation of five bitonic sorting networks S_{2^t}. $m[x]$ represents the value of the bit in position x of the binary representation of m.

Network	Sorting Stages	Implementation directives			Cost	
		Reuse	Ranges	Constraints	Number of 2-input sorters ($w \geq 2$)	2-input sorters used
SN1	$\frac{t}{2}(t+1)$	w	$1 \leq w \leq n$	$w \mid n$	$\frac{w}{4}t(t+1)$	S_2
SN2	$\frac{t}{2}(t+1)$	w,d_i	$1 \leq d_i \leq j$ $1 \leq w \leq n$	$d_i \mid j$ $w \mid n$	$\frac{w}{2}\left(1+\sum_{i=1}^{t-1} d_i\right)$	S_2
SN3	$\frac{t}{2}(t+1)$	w	$1 \leq w \leq n$	$w \mid n$	$\frac{w}{4}t(t+1)$	X_2 (2 states)
SN4	$\frac{t}{2}(t+1)$	w,d_i	$1 \leq d_i \leq j$ $1 \leq w \leq n$	$d_i \mid j$ $w \mid n$	$\frac{w}{2}\left(1+\sum_{i=1}^{t-1} d_i\right)$	X_2 (2 states)
SN5	t^2	w,d,d'	$1 \leq d, d' \leq t$ $1 \leq w \leq n$	$d \mid t, d' \mid t$ $w \mid n$	$\frac{w}{2}dd'$	X_2 (3 states)

Table 2: Implementation characteristics of the different break down rules in Fig. 1 when applying streaming and iterative reuse. The constraint $a|b$ means that a must divide b evenly. Last column: one state means that S_2 is used; otherwise X_2 is used.

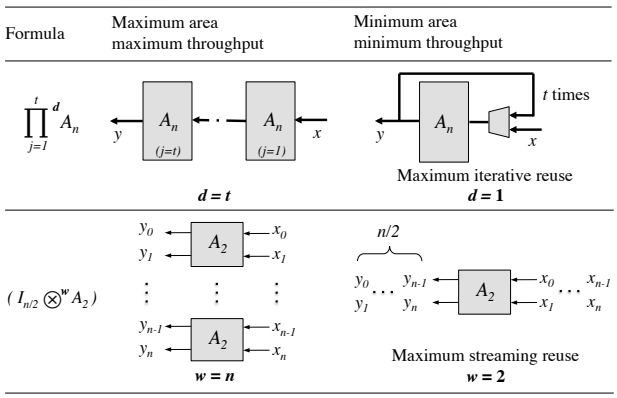

Figure 4: Iterative reuse and streaming reuse applied to expressions.

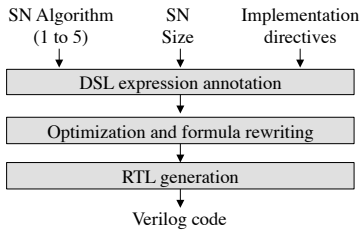

Figure 5: Flow diagram of the generator.

To the best of our knowledge, sorting networks with streaming reuse ($w < n$) have not been implemented before. The likely reason is in the challenge of streaming the required permutations. In our generator we use the method from [14], which uses simple two-ported memories and two-input switches to solve the problem for all permutations in Table 2.

RTL generation. A sorting network expressed in our DSL alongside its implementation directives w and d completely specifies a set of algorithmic and hardware implementation choices. We have created an automated hardware generator (an extension of Spiral [9]) that produces annotated DSL expressions and compiles them to synthesizable register-transfer level Verilog. Fig. 5 illustrates our system's flow.

First, one of the five algorithms in Table 1 is selected and adapted to reflect the user's choices for sorting network size n and implementation directives d and w. The operations for basic blocks S_2 and X_2^c are specified by an intermediate code representation. 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 register-transfer level Verilog. The design is automatically

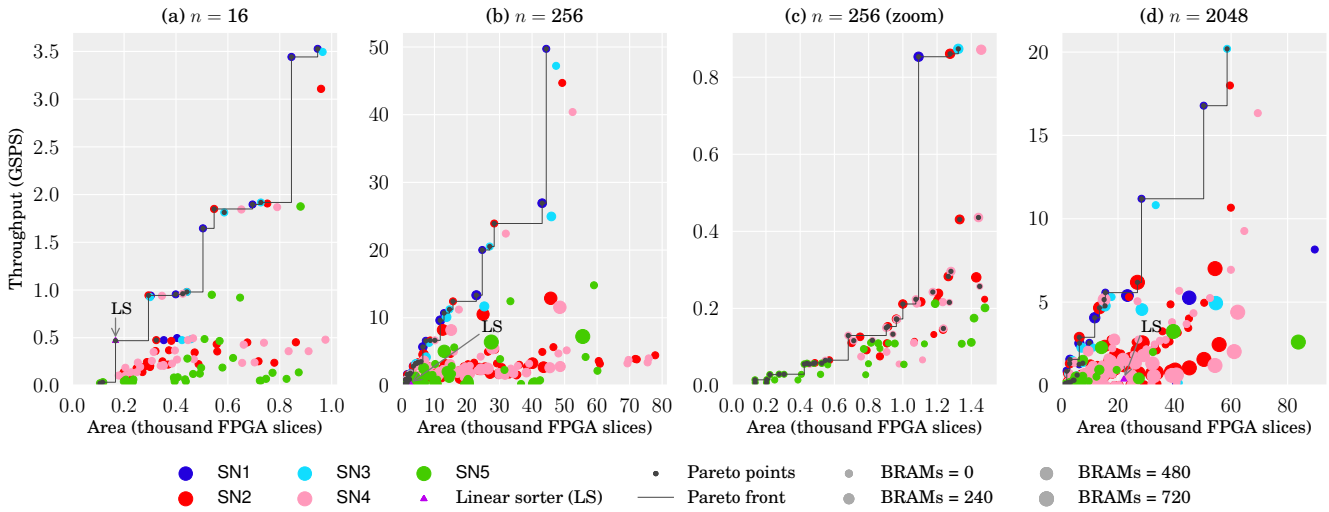

Figure 6: Throughput-area trade-offs for sorting networks of sizes: 16, 256 and 2048.

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

Streaming permutations (where w is less than the number of points permuted) are implemented by the system as explained in [14]. Permutations that operate on non-streaming data (i.e., when all points to be permuted are available in parallel at the same time) are simply implemented with wires.

The generator additionally calculates the system's memory requirements and the latency and throughput of each block. It outputs the final design alongside Verilog testbenches for the verification of the created modules.

4. EVALUATION

With our generator we can systematically explore the large design space spanned by the five different sorting networks and the various reuse options. Each design has a different trade-off between area cost, performance, and memory requirements. In this section we present various experiments on an FPGA for small, medium, and large sizes. We identify Pareto-optimal designs and compare to prior work.

Experimental setup. Verilog descriptions are synthesized and place-and-routed for the Xilinx Virtex-6 XC6VLX760 FPGA using Xilinx ISE 13.1. All designs are generated to process 16-bit fixed-point data. We characterize each design by its cost: hard onchip memory units called Block RAM (BRAM) and the FPGA's reconfigurable slice usage, and by its performance: latency or throughput. The target FPGA includes 720 BRAMs of 36Kbits and 118,000 slices. Latency is measured in microseconds and throughput is measured in giga samples per second (GSPS); both are calculated with the system's maximum execution frequency, given by Xilinx ISE after place-and-route. BRAM and FPGA slice usage are taken from place-and-route reports. An additional parameter to enable or disable the usage of BRAMs was added to our generator, which allows us to further explore platform dependent trade-offs.

Exploring the design space. Fig. 6 shows an evaluation of the design space for the sizes $n = 16, 256, 2048$. All designs that fit onto the target FPGA are shown. 150 designs are shown for $n = 16$, 412 for $n = 256$, and 349 for $n = 2048$. Generating the RTL for each design took a few milliseconds, while obtaining precise resource usage and timing information for each design took from minutes to hours, depending on the complexity of the implementation. In each plot, the x-axis is the number of slices used, the y-axis the throughput, and the size of the marker is proportional to the number of BRAMs used. Fig. 6(c) zooms into a small region of Fig. 6(b).

Among the designs, only the Pareto-optimal ones have to be considered for an application. The (throughput, area)-optimal designs are connected by a line, and all Pareto-optimal (considering area, throughput and BRAM-usage the target objectives) are marked by a black dot. As expected, the smallest design in all cases is obtained with SN5, $w = 2$ and $d = d' = 1$.

Fig. 6(a) shows the solutions found for $n = 16$. All possible designs fit in the target FPGA. The highest throughput is obtained with $w = 16$. Plot (b) shows all the solutions obtained for $n = 256$. The design that achieves best performance is SN1 with $w = 128$. Close to it we find SN2, SN3, SN4, all with $w = 128$ and no iterative reuse, which require more area and achieve lower execution frequencies. Plot (c) shows designs for $n = 256$ that fit in 2000 slices. The smallest designs are SN5, followed by SN2 and SN4. The smallest fully streaming designs, i.e., without iterative reuse, achieve best performance in the range: SN1, SN2, SN3, SN4 all with $w = 2$. Plot (d) shows all the solutions obtained for $n = 2048$. The design that achieves best performance is SN3 with $w = 64$.

For comparison purposes, we added to our design space an implementation (by hand) of a linear sorter, as described in [8, pp. 5]. It is part of the Pareto optimal set only for $n = 16$.

In summary, the set of Pareto-optimal points is non-obvious and composed of different networks and implementation directives. Our generator enables a systematic exploration and identification of these designs. The area requirements of a design could be estimated from the number of 2-input sorters given in Table 2 to guide designers on selecting only a subset of designs for hardware synthesis.

Comparison to prior work. A key contribution is not only the ability to generate designs but, to the best of our knowledge, the first designs for sorting networks with streaming reuse to dramatically reduce area cost, even for high throughput designs. We illustrate this contribution in Fig. 7, which shows the area cost (y-axis) of various sorters for increasing n (x-axis) that fit onto the

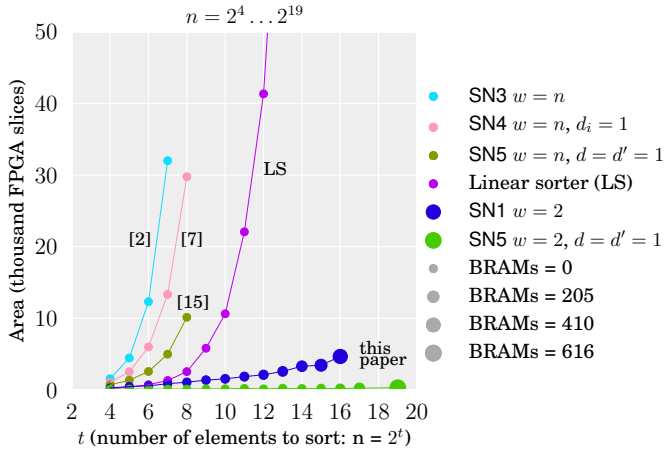

Figure 7: Area usage growth with the number of elements to sort $n = 2^t$.

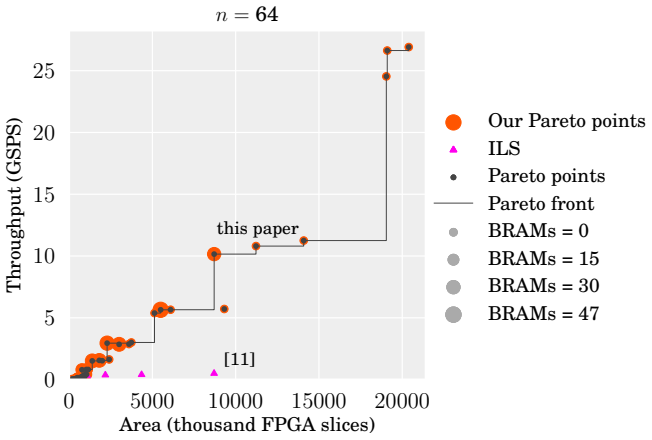

Figure 8: Streaming sorting networks generated in this paper in comparison with the interleaved linear sorters (ILS) from [11].

target FPGA. With prior work it was possible to build the designs of the four uppermost lines: networks without reuse (here: SN3 with $w = n$), iterative reuse in networks with constant geometry ([7] implements SN4 with $d_i = 1$ and [15] implements SN5 with $d = d' = 1$), and the linear sorter (which is not based on a network) from [8, pp. 5].

Our work enables, for example, the two bottom lines: fully streaming designs (here SN1 which processes $w = 2$ elements per cycle) and a design with maximal reuse (SN5 with $w = 2$ and $d = d' = 1$), i.e., it uses only one sorter X_2^c. Both designs require more BRAM but considerably less slices. Even a sorter for 2^{19} elements can be fit onto the target FPGA.

We also compared our designs with the interleaved linear sorters (ILS) presented in [11]. For $n = 64$, Fig. 8 shows throughput and area of our Pareto-optimal designs connected by a line and the ILS designs with $w = 1, 2, 4, 8$ from [11]. Again, the improvement is considerable.

5. CONCLUSIONS

We believe that for well-understood kernel functions IP core generators based on small domain-specific languages offer a practical solution that is hard to match by human designs. We have presented such a generator for sorting. Using the generator, we could generate a large set of candidate designs that capture the existing algorithm knowledge of bitonic sorting networks as well as the various implementation options. The Pareto-optimal designs are non-obvious. Equally important, we used prior work on streaming permutations to present the first sorters with streaming reuse to dramatically reduce area and hence to enable much larger sorting problems to be processed at high throughput on FPGAs.

6. REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) Sorting Network. In *Symposium on Theory of computing (STOC)*, pages 1–9. ACM, 1983.

[2] K. E. Batcher. Sorting Networks and their Applications. In *Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968.

[3] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009.

[4] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. *Circuits, Systems, and Signal Processing*, 9:449–500, 1990.

[5] D. Knuth. *The Art of Computer Programming: Sorting and searching*. The Art of Computer Programming. Addison-Wesley Pub. Co., 1968.

[6] C. Layer and H.-J. Pfleiderer. A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently accessible data. In *Field Programmable Logic and Application*, volume 3203 of *Lecture Notes in Computer Science*, pages 648–657. Springer, 2004.

[7] C. Layer, D. Schaupp, and H.-J. Pfleiderer. Area and Throughput Aware Comparator Networks Optimization for Parallel Data Processing on FPGA. In *Symposium Circuits and Systems (ISCAS)*, pages 405–408, 2007.

[8] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, volume 1. M. Kaufmann Publishers, 1992.

[9] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. Formal Datapath Representation and Manipulation for Implementing DSP Transforms. In *Design Automation Conference (DAC)*, pages 385–390, 2008.

[10] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. In *VLDB Endowment*, volume 2, pages 910–921, 2009.

[11] J. Ortiz and D. Andrews. A Configurable High-throughput Linear Sorter System. In *International Symposium on Parallel Distributed Processing (IPDPSW)*, pages 1–8, 2010.

[12] M. S. Paterson. *Improved Sorting Networks with O (log N) Depth*, volume 5. Springer, 1990.

[13] M. C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM*, 15(2):252–264, 1968.

[14] M. Püschel, P. A. Milder, and J. C. Hoe. Permuting Streaming Data Using RAMs. *Journal of the ACM*, 56(2):10:1–10:34, 2009.

[15] H. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, C-20(2):153 – 161, 1971.

SUPPLEMENTAL MATERIAL

This section includes material that clarifies concepts from Section 2 or results from Section 4.

Fig. 9 shows the data flow graph representation of a bitonic sorter of size $2^3 = 8$ based on SN5. SN5 is a sorting network introduced in Section 2. It is composed of $3^2 = 9$ stages. Each stage performs $8/2 = 4$ parallel compare operations followed by the perfect shuffle permutation.

Table 3 shows FPGA slices and BRAMs used by some of the sorting networks that we generate. The table also includes a linear sorter that was separetaly implemented for comparison. The last design in the table (last 2 rows) are the smallest sorting networks that we generate for a given n. This data was used for the plot in Fig. 7.

Fig. 10 shows the area and latency of designs for size $n = 2048$ that fit into 10,000 slices. The smallest Pareto-optimal designs are provided by SN5 with various degrees of reuse. Most of the low-latency Pareto-optimal designs in the range are found with SN1 and SN3.

Fig. 11 and Fig. 12 zoom into a small region of the plots in Fig. 6(a) and (d) respectively. Both figures display the smallest solutions found in each design space.

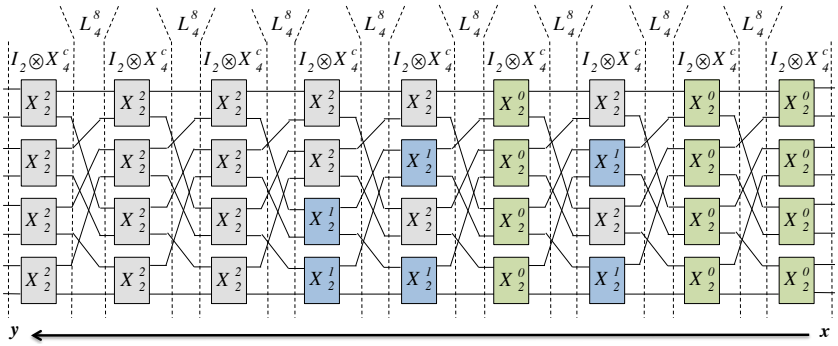

Figure 9: Data flow graph representation of a bitonic sorter of size 8 based on SN5. X_2^0 preserves the original order of the input elements, X_2^1 sorts the inputs in descending order, and X_2^2 sorts the inputs in ascending order.

n	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384
t	4	5	6	7	8	9	10	11	12	13	14
SN3 $w = n$ FPGA slices	1,579	4,476	12,325	32,008							
BRAMs	0	0	0	0							
SN4 $w = n, d_i = 1$											
FPGA slices	1,067	2,546	6,009	13,337	29,776						
BRAMs	0	0	0	0	0						
SN5 $w = n, d = d' = 1$											
FPGA slices	727	1,372	2,591	5,007	10,154						
BRAMs	0	0	0	0	0						
Linear sorter (LS)											
FPGA slices	167	434	749	1,332	2,572	5,840	10,641	22,076	41,327	82,566	
BRAMs	0	0	0	0	0	0	0	0	0	0	
SN1 $w = 2$											
FPGA slices	303	483	582	890	1,092	1,391	1,567	1,876	2,134		
BRAMs	9	14	20	27	35	44	54	67	86	117	172
SN5 $w = 2, d = d' = 1$											
FPGA slices	120	122	124	136	136	161	180	231	159	249	195
BRAMs	1	1	1	1	1	1	1	2	5	10	19

Table 3: Area and BRAM usage for different network sizes n. This data corresponds to the trend lines displayed in Fig. 7.

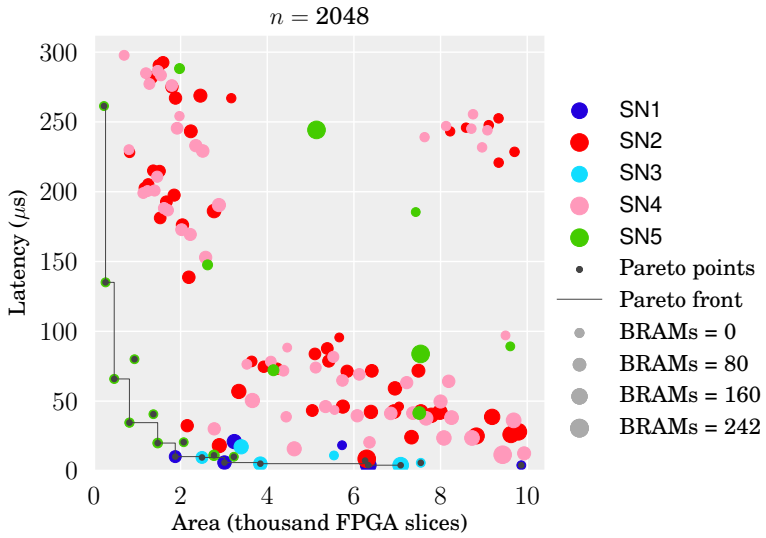

Figure 10: Latency-area trade-offs for sorting networks of size 2048.

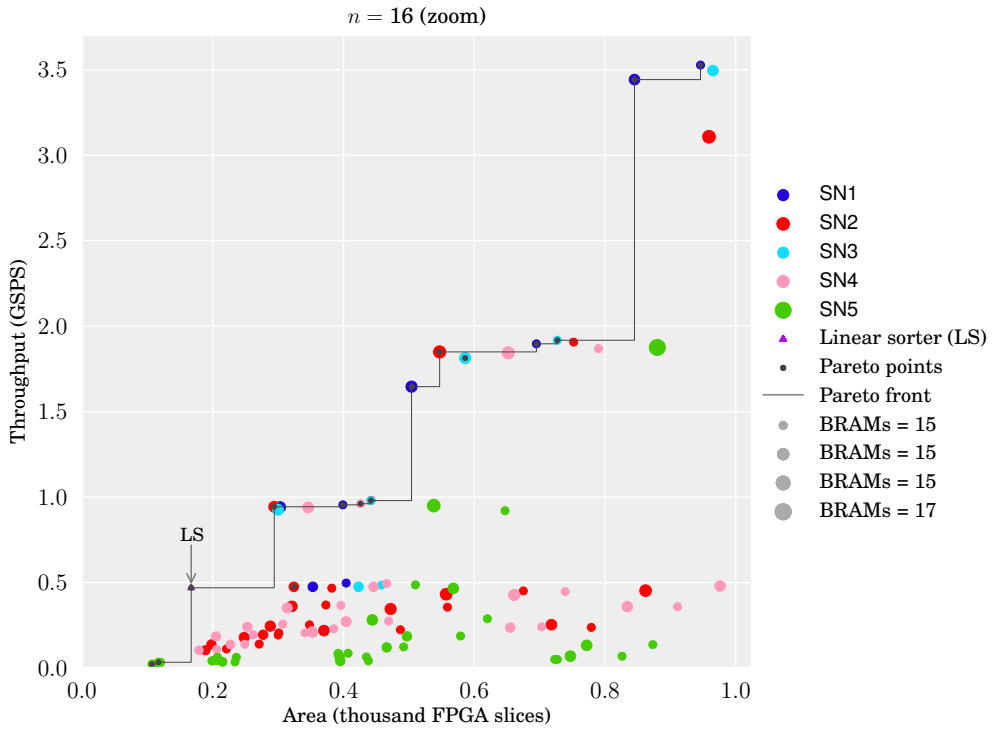

Figure 11: Throughput-area trade-offs for sorting networks of size 16 that fit in 1000 FPGA slices. All the design points are displayed in Fig. 6(a).

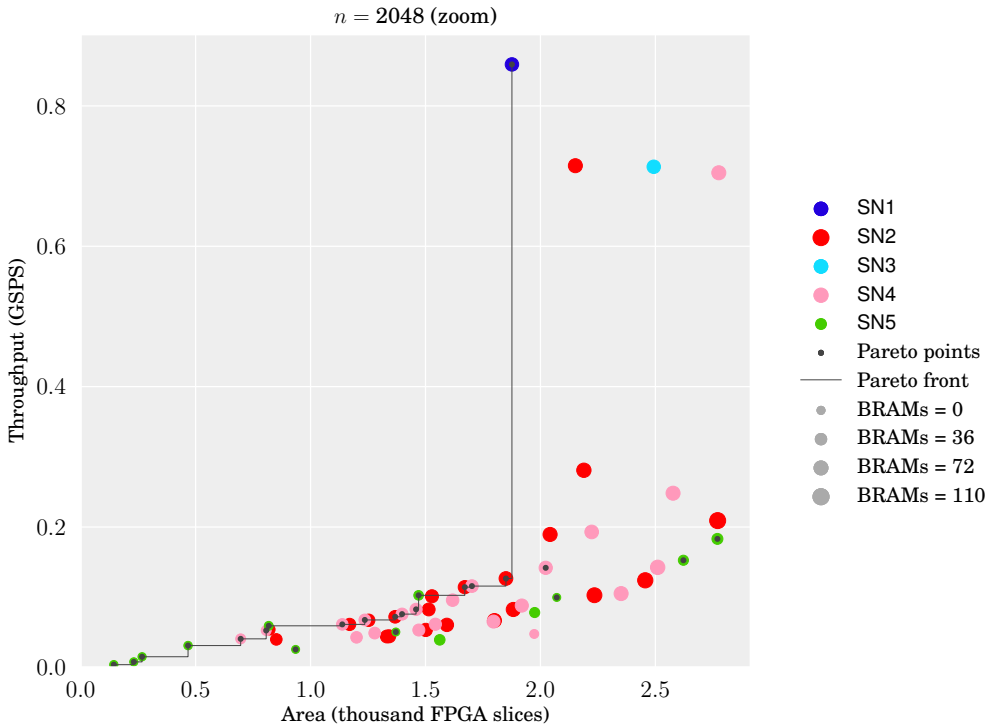

Figure 12: Throughput-area trade-offs for sorting networks of size 2048 that fit in 3000 FPGA slices. All the design points are displayed in Fig. 6(d).