Scotch: Generating FPGA-Accelerators for Sketching at Line Rate

Martin Kiefer
Ilias Poulakis
Sebastian Breß
Technische Universität Berlin
firstname.lastname@tu-berlin.de

Volker Markl
Technische Universität Berlin
German Research Center for Artificial Intelligence (DFKI)
volker.markl@tu-berlin.de

ABSTRACT

Sketching algorithms are a powerful tool for single-pass data summarization. Their numerous applications include approximate query processing, machine learning, and large-scale network monitoring. In the presence of high-bandwidth interconnects or in-memory data, the throughput of summary maintenance over input data becomes the bottleneck. While FPGAs have shown admirable throughput and energy-efficiency for data processing tasks, developing FPGA accelerators requires a sophisticated hardware design and expensive manual tuning by an expert.

We propose Scotch, a novel system for accelerating sketch maintenance using FPGAs. Scotch provides a domain-specific language for the user-friendly, high-level definition of a broad class of sketching algorithms. A code generator performs the heavy-lifting of hardware description, while an auto-tuning algorithm optimizes the summary size. Our evaluation shows that FPGA accelerators generated by Scotch outperform CPU- and GPU-based sketching by up to two orders of magnitude in terms of throughput and up to a factor of five in terms of energy efficiency.

PVLDB Reference Format:
Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. Scotch: Generating FPGA-Accelerators for Sketching at Line Rate. PVLDB, 14(3): 281 - 293, 2021.
doi:10.14778/3430915.3430919

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/martinkiefer/Scotch.

1 INTRODUCTION

Sketching algorithms are a powerful tool for analyzing large data sets and high-velocity data streams [10]. They allow for a trade-off between accuracy and efficiency by constructing lossy but space-efficient summaries in a single pass over the input data. The created summaries enable efficient approximate computation of quantities such as value frequencies or aggregate functions on relational operators. They have been successfully applied to data-intensive tasks such as selectivity estimation [36], heavy hitter and change detection [35], data integration [41], and machine learning [21].

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430919

Since analyses over sketch summaries shift the computational pressure from the analysis to the summary construction, maintaining the summaries at high throughput is critical. Implementations based on multi-core CPUs or GPUs have high energy consumption and often fail to deliver the bandwidth required to satisfy modern interconnects for network (100G Ethernet, Infiniband) and storage (PCIe 3.0+, SATA Express). Field-Programmable Gate Arrays (FPGAs) allow developers to construct custom hardware based on reconfigurable logic elements. Custom hardware allows for high degrees of parallelism, which enables data processing at line rate. These capabilities are a perfect match for the parallel computations over state commonly found in sketching algorithms.

However, implementing sketching algorithms on FPGAs is tedious. An FPGA expert is required to find an implementation that satisfies bandwidth constraints and resource limits while maximizing the sketch size for optimal accuracy. The expert has to make performance-critical design decisions, including memory architecture and pipelining of operations. Furthermore, maximizing the summary size requires time-consuming manual tuning. Previous research in the area focused on implementation strategies for individual sketches and use cases manually tailored to the FPGA [8, 29, 35].

In this work, we take a holistic perspective on FPGA-accelerated sketching by creating FPGA accelerators for an entire class of sketching algorithms without the need for explicit hardware description or manual tuning. We propose the *Scotch* framework that makes four main contributions:

(1) Programming models to describe a variety of different sketching algorithms and ScotchDSL, a domain-specific language to implement user-defined functions for these models
(2) A code generator producing a highly efficient hardware description based on ScotchDSL functions
(3) An auto-tuning algorithm that maximizes the sketch size within the resources of the FPGA and target throughput
(4) An extensive evaluation of our approach on various FPGAs that covers comparisons to CPU and GPU baselines in terms of throughput and energy-efficiency

In the following section, we provide background on sketching algorithms and FPGAs. Section 3 provides an overview of the Scotch system. We then introduce ScotchDSL and its programming model in Section 4. Section 5 explains the code generator and the generated hardware architecture. It is followed by extensions for data parallelism in Section 6 and a discussion of the limitations of the approach in Section 7. Finally, we introduce our auto-tuning algorithm in Section 8. Section 9 presents our experimental evaluation. Section 10 covers related work, and Section 11 concludes the paper by summarizing our findings.

2 BACKGROUND

2.1 Sketching Algorithms

While the term sketching appears in various connotations, it mostly occurs with streaming algorithms that construct a limited size summary in a single pass [10]. Often, sketching algorithms apply updates to the summary state in a randomized fashion by exploiting families of hash functions with statistical properties. The resulting summary allows for the computation of approximate quantities with probabilistic or exact error bounds, like the number of distinct elements [15, 16], distribution characteristics [17], or aggregates over relational operators [11, 36] that would be expensive or infeasible to compute accurately. Supported quantities, their computation, and the error bounds are dependent on the algorithm and hash functions used. The size of the summary balances accuracy, memory, and compute time. Furthermore, sketch summaries constructed over separate data are usually mergeable into a single summary, which allows for parallelization and distributed construction [1].

Sketching algorithms have received much interest from academia and industry. Various sketching algorithms have been proposed and successfully applied in applications, including machine learning [21], data integration [41], approximate query processing [9], change detection [35], and query optimization [36].

Sketching algorithms shift the computational burden from the analysis to the construction of the sketch summary. We will refer to the task of sketch summary construction briefly as *sketching*. Sketching at high throughput and low energy consumption is a crucial problem for improving the efficiency of sketching applications.

2.2 Field-programmable Gate Arrays

A field-programmable gate array (FPGA) is a hardware architecture that allows for the implementation of integrated circuits, similar to ASICs [32]. However, FPGAs support changing the implemented circuit based on software. Thus, FPGAs provide the benefits of custom hardware in terms of parallelism and energy-efficiency while being reprogrammable, making them a promising architecture for accelerating data processing tasks [24, 25, 31, 33, 38, 40].

FPGAs mainly achieve reprogrammability through hundreds of thousands to millions of programmable look-up tables (LUTs) spread throughout the fabric, each representing a boolean function with a fixed number of inputs. More complex functions are constructed by connecting LUTs via a configurable routing network. Furthermore, Block RAM (BRAM) elements provide random memory in the order of kilobits with configurable width and depth. The availability of LUTs and BRAM elements determines the maximum complexity of the implemented logic.

The maximum clock rate is a critical factor for the throughput achieved by FPGA designs, which has to match the interconnect used to interface the FPGA with I/O. The signal delay caused by logic and routing between registers determines the maximum clock rate. Thus, pipelining computations by adding intermediate registers is essential to achieve high throughput. To allow for heavily pipelined designs, FPGAs provide groups of LUTs accompanied by D-Flipflops. We refer to these groups as elementary logic units (ELUs) [32]. FPGAs are commonly programmed using the Register-Transfer-Level (RTL) abstraction provided by VHDL or Verilog, which describes the flow of data between registers.

3 SYSTEM ARCHITECTURE

In this section, we provide an overview of Scotch and the accelerators generated. In Section 3.1, we motivate and introduce the design requirements of Scotch. Next, Section 3.2 shows the architecture of the accelerators generated by Scotch. Finally, Section 3.3 gives a high-level overview of the Scotch system.

3.1 Design Requirements

In the following, we describe the problems and highlight the design requirements that are the foundation of Scotch.

DR1: Lightweight Sketch Specification. While sketching is typically very concise in its mathematical definition, implementing it on an FPGA adds additional complexity. An FPGA expert is required since the developer needs hardware design knowledge to make architectural decisions. In particular, the expert has to pipeline computations and decide on a memory architecture. Scotch provides an intuitive programming model and domain-specific language to describe sketching. These descriptions are concise and close to their mathematical definition. Code generation replaces the tedious process of programming RTL for these functions. Efficient auxiliary components, such as memory, are generated automatically according to the desired sketch summary size without user involvement.

DR2: Automated Tuning. As a sketch summary's accuracy increases with its size, providing a large summary size is crucial for a sketching accelerator. However, finding a large sketch size within the FPGA's resources while meeting the operating frequency required by the interconnect is tedious. The developer has to vary the size and compile the accelerator by trial-and-error. This process requires an FPGA expert's intuition and is inconvenient given compile times in the order of hours. Scotch maximizes the sketch size for a given FPGA and interconnect without manual tuning. An auto-tuning algorithm systematically varies the summary size while being economical in the number of performed compilations.

DR3: Device and I/O Agnosticism. Various FPGAs and boards exist with different supported interconnects and target domains ranging from IoT applications to large-scale network processing. Implementations created for a particular setup are usually not easily portable to another. Scotch separates the implemented algorithm from the FPGA vendor, model, and board by encapsulating these details in an I/O module. RTL generation and tuning make no assumptions on the device or interconnect.

DR4: Constant Processing Rates. Compared to general-purpose processor architectures such as GPUs or CPUs, constructing custom hardware on FPGAs has a significant benefit: They can provide high throughput at a constant rate, which enables data processing at the full rate of the interconnect. However, this requires a careful design of all components implementing the sketching functionality. Scotch generates hardware that processes data at a constant rate. All components are scalable with the sketch size and fully pipelined, meaning that all components are divided into pipelined substages and process one set of input values per clock cycle. They do not require stalls or flushing pipelines due to data dependencies. For high-throughput interconnects, Scotch provides mechanisms to exploit data parallelism.

3.2 Accelerator Design

The accelerators generated by Scotch consist of two components connected via a common interface:

Sketching Unit: Processes input values by storing and manipulating the sketch summary state. Furthermore, it exposes sketch summary state when requested via control signals. Scotch generates and optimizes the sketching unit.

I/O Controller: Implements off-chip communication via an interconnect, such as Ethernet or PCIe. It interprets signals arriving on the interconnect as input values or as requests to expose the sketch summary and drives the interface signals accordingly. The I/O controller has to be provided by an expert as, similar to drivers in operating systems, its implementation depends heavily on the used FPGA, board, and interconnect.

The separation of sketching and I/O enables flexibility in terms of the interconnect (DR3).

3.3 Scotch

Scotch generates optimized hardware accelerators for a broad class of sketching algorithms. Users specify the sketching process in a convenient domain-specific language, while code generation and automated tuning replace the complicated and time-consuming RTL development and manual tuning of the summary size. Figure 1 illustrates the high-level system architecture.

A user implements sketching algorithms by providing user-defined first-order functions. They are arguments to higher-order functions supported by Scotch. User-defined functions are given in *ScotchDSL*, a domain-specific language that allows for an intuitive description of the sketching computations close to their mathematical formulation. ScotchDSL and the underlying programming model satisfy DR1 and are described in Section 4.

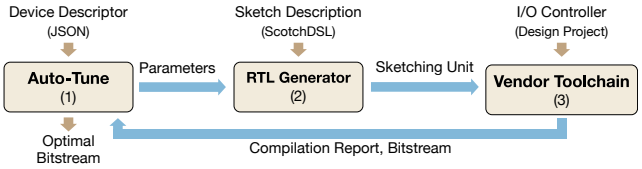


Figure 1: Scotch system architecture

The *RTL generator* produces hardware description code for the entire sketching unit based on ScotchDSL functions. The generated RTL is fully pipelined, enabling high constant processing rates (DR4). The RTL generator and the generated hardware architecture are discussed in Section 5. A top-level project, which contains the I/O controller and constraints, instantiates and connects to the generated sketching unit yielding a complete design for the accelerator. Finally, the vendor-specific toolchain compiles the design, resulting in a bitstream to configure the FPGA and reports on resource utilization and timing.

Exploiting data parallelism is a common technique to achieve high throughput in parallel processor architectures. The RTL generator supports data parallelism by trading FPGA resources for higher maximum throughput (DR4). We discuss the underlying approaches in Section 6.

Scotch's *auto-tuning* algorithm maximizes the target sketch size within the resource limitations of the FPGA and clock rate constraints set by the I/O controller for interconnect used. Thus, it ensures high accuracy while creating a fully functional accelerator. The algorithm repeatedly parameterizes the RTL generator, compiles the project, and analyzes the resource consumption and timing reports in a feedback loop. The auto-tuning algorithm satisfies DR3 and is given in Section 8.

4 SKETCH SPECIFICATION

In this section, we introduce the sketch specification approach used in Scotch to satisfy DR1. In Section 4.1, we propose the *Select-Update model* that allows for a convenient description of sketching in terms of user-defined functions. It serves as the underlying programming model for Scotch. In Section 4.2, we introduce ScotchDSL, a domain-specific language allowing developers to specify these user-defined functions close to their mathematical formulation while being translatable to RTL.

4.1 Select-Update Model

The maintenance process of many popular sketching algorithms can be generalized as updating one entry per row of a matrix. Based on this observation, we propose the Select-Update model to describe sketching by specifying a *select function* that selects the entry in a row and an *update function* that determines the new value of the selected entry. Table 1 provides a non-exhaustive list of sketching algorithms fitting this model. The Select-Update model serves as a programming model for Scotch.

Formally, the Select-Update model defines an update to the sketch matrix as follows: A sketch matrix $S \in \mathbb{S}^{m \times n}$ is adapted for each observation $t \in \mathbb{T}$. For each row $i \in \{1 \dots m\}$, a selector function $sel_i : \mathbb{T} \to \{1, \dots, n\}$ determines an entry that is updated based on an update function $up_i : \mathbb{T} \times \mathbb{S} \to \mathbb{S}$. Formally, an update is defined as:

$$S\left[i, sel_i(t)\right] := up_i\left(t, S\left[i, sel_i(t)\right]\right), \quad i \in \{1 \dots m\} \tag{1}$$

The domains of the state $\mathbb{S}$ and value $\mathbb{T}$ are fixed-size bit sequences; their interpretation is left to the select and update function.

Select and update functions are either the same for all i or drawn from a family of functions based on a randomly drawn seed $\theta \in \Theta$, where the seed domain Θ is a fixed-size bit sequence as well. This allows for convenient definitions in a single function that takes the seed as an optionally used third argument. Thus, we specify $sel_i(t)$ and $up_i(t, s)$ by specifying $sel(t, \theta_i)$ and $up(t, s, \theta_i)$, respectively.

The Select-Update model includes sketching algorithms operating on a single column, such as AGMS [2] or MinHash [4]. In these cases, the specification of a select function is unnecessary, as there is only a single entry per row. This property allows for simplifications and optimizations, which we highlight throughout the paper. We refer to these sketches as *column sketches*, while referring to the general case as *matrix sketches*. Sketches that operate on a single row are referred to as *row sketches*.

We provide an example for a matrix sketch and a column sketch in the Select-Update model.

Example 1 (Count-Min): The Count-Min sketch [11] is a popular algorithm that provides an upper bound on the frequency of

observed elements and other quantities such as join sizes. First, a member of a family of two-wise independent hash functions computes the offset in each row. A common choice is *H3* [27], which computes an h-bit hash value for a k-bit key by using a random seed θ consisting of $h \cdot k$ bits.

$$sel(t, \theta) - 1 = \bigoplus_{j \in \{0 \ldots k-1\}} (t\,[j] \wedge \theta\,[j \cdot h + 1 \ldots (j+1) \cdot h]) \quad (2)$$

The subtraction in the first term accounts for one-based indexing. The operator $\oplus$ denotes a sequential bitwise XOR operation; the operator $\wedge$ denotes a bitwise logical AND between the j-th bit of t and j-th sequence of h bit in θ. The following update function denotes an increment to the selected state:

$$u(t, s, \theta) = s + 1 \quad (3)$$

Example 2 (AGMS): The AGMS sketch maintains a column vector as state and allows estimating join sizes and other relational aggregates [2, 36]. A family of independent hash functions maps a k-bit key to $\{+1/-1\}$ updates applied to each counter. The *EH3* family was found to be a good choice [13, 28]. The EH3 hash functions require a random seed θ consisting of $k + 1$ bits.

The function *eh*3 applies bitwise operations on the input value and the seed:

$$eh3(t, \theta) = h(t) \oplus \theta[k+1] \oplus \bigoplus_{r \in \{1 \ldots k\}} (\theta\,[r] \wedge t\,[r]) \quad (4)$$

$$h(t) = \bigoplus_{r \in \{1 .. \frac{k}{2}\}} t[2r-1] \vee t[2r] \quad (5)$$

The update function is then defined as:

$$u(t, s, \theta) = s + \begin{cases} 1 & \text{if } eh3(t, \theta) = 1 \\ -1 & \text{else} \end{cases} \quad (6)$$

Table 1: Sketching algorithms generalized by the Select-Update model and implementable in ScotchDSL

FM [16]	MinHash [4]	Fast-AGMS [9]	Count-Min [11]
AGMS [2]	HyperLogLog [15]	Bloom Filter [3]	Fast-Count [34]

4.2 ScotchDSL

Scotch generates the sketching unit RTL based on user-defined select and update functions. A programmer provides these functions in ScotchDSL, a domain-specific language that describes the flow of computations from the input variables to the function result in terms of operations on bit vectors of arbitrary size. It is sufficiently expressive to implement the sketching behavior while ensuring that the computations translate to efficient hardware.

An algorithm implementation in ScotchDSL consists of an implementation of the user-defined functions and a descriptor file. The descriptor file contains the number of bits required for the state, input value, seeds, and auxiliary variables. ScotchDSL function implementations consist of consecutive variable assignments that set the value of a variable to the result of the expression on its right-hand side. A ScotchDSL function ends with an assignment to the output variable.

Listing 1: Select function for Count-Min with H3

```
1  x <= expand(t(0),32) & seed(31 downto 0) ;
2  for i in 1 to 30 {
3        x <= (expand(t(i),32) & seed((i+1)*32-1 downto i*32)) ^ x;
4  }
5  offset <= (expand(t(31),32) & seed(32*32-1 downto 31*32)) ^ x ;
```

ScotchDSL supports the following operations in the expressions:

(1) Selecting an individual bit or a range of bits from a bit vector
(2) Bitwise logical operations and comparisons
(3) Signed and unsigned arithmetic operations and comparisons
(4) Auxiliary functions that take variables as an argument and return a bit vector as a result.

Besides simple assignments, we support conditional assignments and for-loops. For-loops have a fixed iteration range and replace repetitive assignments with a fixed pattern.

Operating on bit vectors allows the specification of algorithms closer to their mathematical definition (DR1). For example, the EH3 hash function given in Equation 4 requires computations on a 33-bit seed, which complicates an implementation in programming languages such as C/C++, as only word-sized data types are provided. ScotchDSL functions usually consist of few lines and allow for quick customizations such as changing the state size or adding a frequency to an update.

As hardware definition languages operate on bit vectors in the same way, all expressions map to equivalent expressions in the target language VHDL. By providing a restricted set of operations, we ensure that the function code maps to pipelined RTL that is synthesizable on an FPGA (Section 5.2). Constructs that prevent a fully-pipelined design or are not synthesizable to FPGA hardware are not supported (e.g., data-dependent loops).

The ScotchDSL syntax borrows from VHDL. In the following, we provide ScotchDSL implementations for the previously introduced algorithms and highlight the language constructs. The descriptor files are regular JSON files and omitted for the sake of space. Note that indexes in ScotchDSL are zero-based.

Example 1 (Count-Min, Select): Listing 1 shows the implementation of the Count-Min/H3 select function given in Equation 2. The code computes a 32-bit hash value (*offset*) from a 32-bit observation (t) based on a $32 \cdot 32 = 1024$-bit seed (*seed*).

Line 1 shows the regular assignment of a vector expression to a variable. It computes the first iteration of the sequential XOR in Equation 2. The expression *seed(31 downto 0)* selects the first 32 bits of the seed, and the & operator represents a bitwise AND. The built-in auxiliary function *expand(t(0), 32)* returns a bit vector consisting of 32 bits all set to the zeroth Bit of *t*. The output of the expression is stored in the auxiliary variable *x*. Lines 2-4 show a for-loop. It iterates from 1 to 30 using the variable *i*. The loop body computes the sequential XOR up to the *i*-th bit and seed by applying the XOR operator ^ to the last value of *x*. Line 5 computes the last iteration of the sequential XOR and stores the result in the output variable *offset*.

Example 2 (AGMS, Update): Listing 2 shows the implementation of the update function for AGMS with a 32-bit input value and state. Line 1 computes the result of the sequential XOR operation in Equation 4. It computes the bitwise AND operation between the first 32-bits of the input variable *seed* and the input value *t*. Finally, it

Listing 2: Update function for AGMS with EH3

```
1  p <= parity(seed(31 downto 0) & t);
2  h <= parity((t(30 downto 0) | t(31 downto 1))
3                          & '10101010101010101010101010101');
4  outstate <= p ^ h ^ seed(32) = '1' ?
5                          signed(state)+1 : signed(state)−1;
```

computes the full sequential XOR by using the *parity* auxiliary function provided by Scotch. Similarly, Line 2-3 computes the non-linear function h given in Equation 5. It computes a bitwise OR between the first and last 31 bits of the input value v. The parity function computes the sequential XOR; the bit vector literal in Line 3 ensures that only disjoint pairs of bits contribute to the parity. Finally, Lines 4-5 compute a $+1/-1$ update as shown in Equation 6 by using a conditional assignment of the form *var <= condition ? expr : expr*. In the condition, we compute the full result of the function $eh3$ given in Equation 4 and check whether the result is equal to the bit vector literal *'1'*. If the condition holds, the current state is incremented; otherwise, it is decremented. The built-in function *signed* assigns signed integer semantics to a bit vector for arithmetic.

5 RTL GENERATOR

In this section, we introduce our approach to RTL generation for the sketching unit. In Section 5.1, we provide an overview of the RTL generator and the sketching unit architecture. Section 5.2 describes the translation of ScotchDSL functions to function units that perform all algorithm-specific computations. A pipelined RAM holds the summary state. We explain its architecture in Section 5.3. Compute units contain the ScotchDSL function units and perform all auxiliary operations. We explain the the compute unit architecture and its components in Section 5.4. The state transfer controller retrieves and exposes the sketch state. We describe its architecture in Section 5.5.

5.1 Overview

The RTL generator creates a VHDL hardware description for the sketching unit based on ScotchDSL functions. The number of rows and columns for the sketch are input parameters and varied by Scotch's auto-tune algorithm. According to the desired state matrix shape, the RTL generator instantiates and parameterizes all components in the sketching unit. We provide an overview of the sketching unit architecture and then outline the RTL generation process.

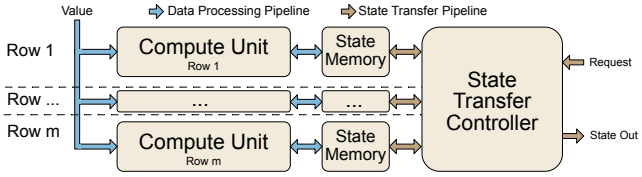


Figure 2: Sketching unit architecture

Figure 2 shows the top-level architecture of the sketching unit. The sketching unit performs two tasks: First, it adjusts the summary state according to input values and, second, exposes it to the I/O controller. Each row of the sketch is represented by a dedicated compute unit and state memory, which operate independently and in parallel. The compute unit processes one input value per clock cycle and initiates read and write operations on the state memory. The state transfer controller exposes the sketch state when triggered by an outside request. It connects to the state memory of every row and dispatches read requests.

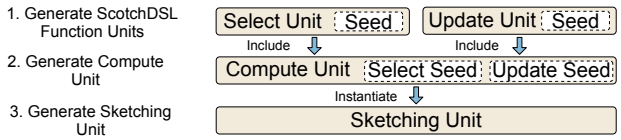


Figure 3: RTL generation process

The RTL generator creates the sketching unit's hardware description in a three-step process shown in Figure 3. First, the generator translates ScotchDSL to function units. They implement the computation of the select and update functions while leaving the random seed as an open parameter. Second, it generates a compute unit by adding all auxiliary components. Third, the RTL generator creates the sketching unit. It instantiates a compute unit and state memory, sets the seeds, and adds the state transfer controller.

The sketching unit is fully pipelined to achieve high operating frequencies. The generator adjusts the interfaces and internals automatically according to the size of the state and input value of ScotchDSL functions. In the following, we detail the architecture and generation of the individual components. We assume sketching units consuming one input value per clock cycle, while Section 6 discusses mechanisms to lift this assumption.

5.2 ScotchDSL Function Units

ScotchDSL function units implement the sketch-specific computations in hardware: Select function units compute a row offset from the input value. Update function units compute the new state from the input value and previous state. The RTL generator implements them based on the provided ScotchDSL functions, for which we solve two problems: First, we have to translate the imperative ScotchDSL user-defined functions to pipelined RTL. Second, we have to ensure that the generated RTL does not introduce data hazards inside the update function unit.

The general translation mechanism consists of three steps:

Step 1: Abstract Syntax Tree (AST). The RTL generator parses the input function file based and creates an AST.

Step 2: Dependency Graph. The RTL generator transforms the tree into a dependency graph that contains a node for every function input variable and assignment. It unrolls loops in the process. When the statement node B directly depends on the result of a node A, a directed edge from a node A to node B is inserted. The resulting dependency graph represents the flow of computations from input variables to the output variable.

Step 3: Function Unit. The RTL generator translates the assignment graph to RTL for the function unit. Each assignment node results in a synchronous component that computes the result and buffers the output. Edges between assignment nodes create connections in the top-level function unit. If necessary, the RTL generator adds buffering to ensure intermediates computed for the same input value arrive at the same clock cycle.

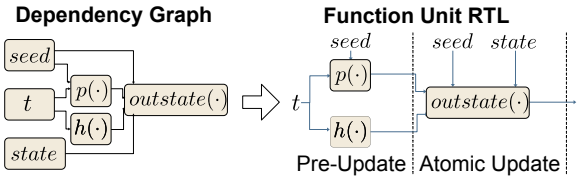


Figure 4: Dependency graph and function unit RTL generated for the AGMS update function (Listing 2)

The resulting function unit computes the ScotchDSL function in a pipelined fashion based on the input value and seed. As our architecture instantiates one dedicated compute unit per row, seeds are constants. Figure 4 shows the dependency graph and the function unit for the AGMS update function given in Listing 2.

While the general translation mechanism is sufficient for select functions, update functions also depend on the previous state. As the update function computes new state values in every clock cycle, data hazards occur if a later update overwrites a state used for a computation in the update function pipeline. Resolving these data hazards requires stalling or flushing the pipeline, which conflicts with DR4. Instead, the RTL generator prevents data hazards by ensuring only the output value computation accesses the previous state. If the function code violates this condition, the RTL generator inlines the computations for assignments required by the output value computation until the condition holds. We refer to the single computation of the new state as the *atomic update* while calling the rest of the update function pipeline the *pre-update*.

5.3 State Memory

The state memory holds each row's state entries in a BRAM-based pipelined random access memory architecture. A modern FPGA contains hundreds to thousands of BRAM blocks, each providing dense random memory in the order of kbits with configurable width and depth. BRAM blocks are dual-ported and process exactly one read and write operation per clock cycle to independent offsets when operating in *simple dual-port mode*. Maintaining the sketch size for large rows necessitates combining multiple BRAM blocks to provide sufficient memory depth.

While synthesis tools can automatically construct deeper RAM by combining k BRAM blocks, they naively multiplex and demultiplex reads and writes. This is feasible for small values of k, however, prior work confirmed that this approach scales poorly compared to a pipelined memory architecture [35].

Figure 5 shows the pipelined memory architecture. It consists of k banks, where each bank contains a BRAM block and is exclusively responsible for a part of the address range. Read and write requests to an address are served by their corresponding memory bank. All other banks forward the request. Memory banks buffer requests before forwarding them to the next bank, creating a pipeline that consumes one read and write request per clock cycle with a latency of k clock cycles and a throughput of one read and write request per clock cycle. The width of the memory is the state size. The bank depth is a parameter to the code generator. Scotch sets it according to the depth supported by the target device's BRAM elements for the given state width.

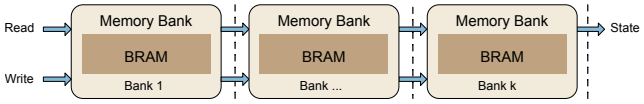


Figure 5: Pipelined memory architecture

The RTL generator simplifies the memory architecture for column sketches to a single register since random memory access is not required.

5.4 Compute Unit

Compute units process the input values for one row of the sketch by evaluating the select and update function and updating the per-row state according to the ScotchDSL function units. .

5.4.1 Overview. The compute units are pipelined and consist of several substages. Primarily, they consist of stages for the select and update function evaluation and stages accessing state memory. Furthermore, the RTL generator adds auxiliary stages to truncate the select function's output value and to prevent data hazards. Figure 6 shows the the compute unit architecture with all substages. Note that the select, pre-update, data forwarding unit (DFU), and memory stages consist of several substages.

Select: The select function unit computes the output of the select function given in ScotchDSL. As Scotch varies the number of columns in the auto-tune algorithm, we assume the select function provides sufficiently large offsets and truncate them to the required range in the truncate stage.

Truncate: We truncate the offset provided by the select function to the range $[0, n-1]$, n being the number of columns in the sketch.

Memory Read: The state memory retrieves the state for the previously computed offset.

DFU: As the compute unit consists of several substages with multiple clock cycles of latency, a state value read from memory for a particular offset can be overwritten by updates further down the pipeline. To prevent data hazards from causing lost updates, a data forwarding unit (DFU) tracks recent updates and ensures that only the most recent state values enter the atomic update stage. Section 5.4.2 introduces our novel fully-pipelined data forwarding unit architecture.

Pre-Update: A concurrent stage computes the inputs to the atomic update stage as these computations do not depend on the state.

Atomic Update: The atomic update is computed based on the intermediates from the pre-update stage and the most recent state value arriving from the DFU. As the DFU can not see the very last update, the atomic update stage tracks its last computed state and uses it in case of two consecutive updates to the same offset.

Memory Write: The state memory stores the previously computed new state.

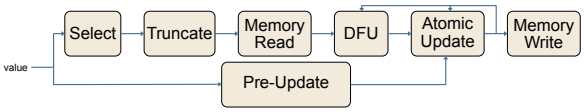


Figure 6: Compute unit architecture

The RTL generator omits the select, truncate, and DFU stages as an optimization for column sketches since no random memory access is required.

5.4.2 Data Forwarding Unit (DFU). The DFU resolves data hazards caused by the pipelined memory architecture. Data hazards occur when a state read from memory is altered by an update further down the pipeline. The DFU delays the computation of the atomic update to replace outdated states with the most recent value.

Figure 7 shows the architecture of our DFU. We first explain the key idea of our DFU and then introduce the optimized architecture used by the RTL generator.

Key Idea: Our DFU tracks the state-offset pairs leaving the atomic update stage in a shift register of size l. It compares state-offset pairs from the memory read stage to the shift register values in l stages. Each stage $i \in \{1 \dots l-1\}$ compares the incoming state to the i-th and $(i+1)$-th least recent entry in the shift register. If one or both entries from the shift register coincide with the input offset, the most recent state in the shift register entries is passed to the next stage instead of the input state (3-way Compare-Forward). The last stage l only performs a single comparison with the most recent entry in the shift register (2-way Compare-Forward). This approach ensures that, when leaving the DFU, a state-offset-pair has observed all updates caused by its second to $2l$-th successors in the pipeline.

Optimization: As 3-way Compare-Forwards are complex and reduce the scalability of the DFU, Scotch avoids them by splitting the DFU stages into three parallel pipelines. The i-th stage of the upper pipeline compares the input value to the i-th value in the shift register, while the lower stage compares the input value to the $(i+1)$-th value. This separation allows us to use simple 2-way Compare-Forward logic. However, the DFU has to track which pipeline carries the most recent state. If there is a match in the lower or upper pipeline exclusively, this pipeline carries the most recent state. If there is a match in both, the $(i+1)$-th value takes priority as it is more recent. In case there was no match, the previous priority remains valid. We perform these computations in a separate priority resolution pipeline with access to the results of the previous stage's comparisons. The final stage forwards the most recent state from either the l-th position in the shift register, the upper pipeline, or the lower pipeline. This requires a 2-way Forward based on the priority and an additional 2-way Compare-Forward.

The size l of the DFU that prevents all data hazards depends on the number of banks in the state memory k. There are a total of $k+l$ successors in the pipeline that may cause data hazards. The DFU and atomic update track the next $2l$ updates. Thus, picking $l = k$ results in a minimal DFU that prevents all data hazards.

5.5 State Transfer Controller

The state transfer controller exposes the state of the sketch summary to the I/O controller. It connects to the state memory of all rows and sequentially reads all $m \cdot n$ state values. As Scotch aims to maximize the sketch size, the state transfer controller must scale with the number of rows. Particularly column sketches require state transfer controllers with a high number of connections.

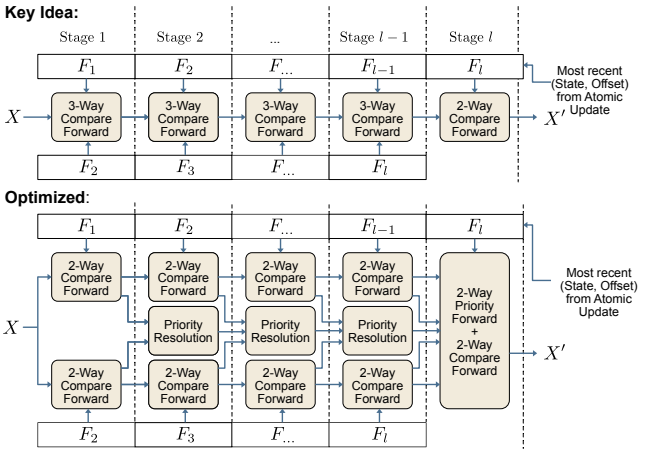


Figure 7: Data forwarding unit architecture

Figure 8 shows the architecture of our transfer controller. A dispatch unit issues read and write requests to the per-row state memory, while a collect unit routes the result of the request to the output signals. Dispatch and collect units both follow a tree-shaped structure where each tree level is a pipeline stage. The pipelined tree structure prevents drops in the operating frequencies due to high fan-out in the dispatch unit and high fan-in in the collect unit by distributing the logic over several stages.

The dispatch unit maintains a counter for rows and offsets. The counters adjust after every request. When the I/O controller requests the next state value, several stages of 4-way dispatch logic (4D) route the selected offset to the selected row's state memory. As soon as the state memory serves the read request, 4-way collect (4C) logic routes the state value to the output signals.

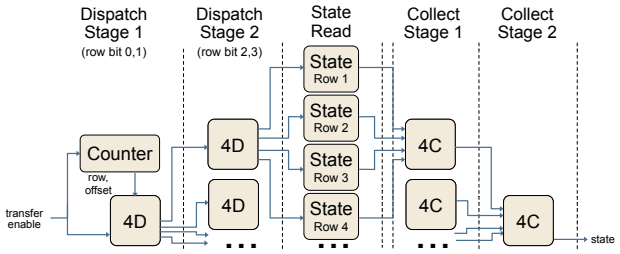


Figure 8: State transfer controller architecture

As the per-row state memory has only one read port, value processing and state transfer are mutually exclusive. Thus, state transfer must wait until all updates are written to state memory.

6 DATA PARALLELISM

The sketching unit described in the previous sections consumes one input element per clock cycle. However, this is not sufficient to satisfy high-bandwidth interconnects such as 100G Ethernet. I/O modules interfacing such interconnects have to forward multiple input elements per clock cycle as the operating frequency of state-of-the-art FPGAs is limited to hundreds of Megahertz. Thus, data-parallel sketching units are required. This section describes the two mechanisms Scotch uses to provide data parallelism and satisfy DR4 for high-bandwidth interconnects: Section 6.1 introduces a general

mechanism that exploits data parallelism by replicating components of the sketching unit. Section 6.2 introduces the *Map-Apply model* as an alternative programming model for column sketches, which explicitly incorporates data parallelism to provide an improved resource utilization.

6.1 Replication

The most intuitive approach to data-parallel sketching with d simultaneous input values is maintaining d replicas of the sketch in parallel. The d separate sketches can be evaluated separately or merged into a single summary by an application. While the approach is simple, it also comes with approximately d times ELU and BRAM consumption.

If a data-parallel sketching unit with a replication factor of d is requested, Scotch generates d replicas of each compute unit and state memory. The components belonging to the same replica connect to the same input value pipeline. The state transfer controller is shared among replicas to save resources. The I/O controller may also instantiate the sketching unit several times to maintain entirely independent replicas and allow symmetric throughput for data processing and state transfer.

6.2 Map-Apply Model for Column Sketches

The Map-Apply model is an alternative model for column sketches that provides better resource utilization than plain replication. It is as expressive as the Select-Update model, but it requires explicitly modeling data parallelism by defining a function that merges the update for d input values before applying the aggregated update.

Intuitively, a *map function* computes intermediates for all d inputs, while an *apply function* merges the intermediates into a single update to the state. Formally, for each row i, a map function $map_i : \mathbb{T} \to \mathbb{X}$ is applied to each of the d input values. A function $ap : \mathbb{S} \times \mathbb{X}^d$ computes the update to the state from the intermediates:

$$S[i] := ap(S[i], map_i(t_1), \ldots, map_i(t_d)),\ i \in \{1 \ldots m\} \quad (7)$$

As for Select-Update, the functions $map_i(t)$ are specified by providing $map(t, \theta_i)$ that may depend on a random seed $\theta_i \in \Theta$.

Example (AGMS): We provide an example for AGMS in the Map-Apply model with d inputs. The map function computes EH3 for an input value and translates the output to $+1/-1$:

$$map(t, \theta) = \begin{cases} 1 & \text{if } eh3(t, \theta) = 1 \\ -1 & \text{else} \end{cases} \quad (8)$$

Finally, the apply function accumulates the d $+1/-1$ updates and adds the update to the previous state:

$$ap(s, t_1, \ldots, t_d) = s + \sum_{r \in \{1, \ldots, d\}} t_r \quad (9)$$

Compute Unit Architecture: The RTL generated for data-parallel sketches described with the Map-Apply model provides a more favorable resource trade-off than replication. First, map function units compute the individual outputs for the d input values. Then, the apply function merges them before computing the new state. As the Map-Apply model reduces all parallel computations to a single update to the state, the number of state registers and the state transfer controller remain independent of d.

7 DISCUSSION

The RTL generator relies on the structure implied by the Select-Update and Map-Apply model to generate fully-pipelined RTL (DR4). However, the underlying models and ScotchDSL also impose limitations on the supported algorithms. In the following, we will discuss these limitations.

Models: The supported models fix the memory access pattern for all supported algorithms, and the per-row memory is the only mechanism to store state. Each input value results in exactly one read and write operation to the state memory. This memory access pattern allows us to use a pipelined memory architecture that handles one read and write operation per clock cycle and resolves data hazards. Lifting this restriction would require us to stall the pipeline during conflicting memory operations and, thus, violate DR4. In particular, the supported models exclude streaming algorithms that require keeping a sorted list of values or complex data structures, such as Space-Saving [23] or Exponential Count-Min [26]. Furthermore, algorithms that require an evaluation of the sketch to perform an update are not supported (e.g., CM-CU [18]).

ScotchDSL: ScotchDSL prevents for-loops with runtime dependent conditions. Supporting this would require us to build hardware that stalls while the loop iterates and, thus, violate DR4. Furthermore, ScotchDSL does not support floating point interpretations of bit vectors. Hardware support for floating-point operations is highly dependent on the device and requires vendor-specific modules that are hard to parameterize automatically. However, this would allow for floating-point state in already supported sketches and enable new sketches, such as Quantile sketches [17] or DDSketch [22]. Thus, we consider this an interesting direction for future work.

There is a well-known trade-off between flexibility, throughput, and resource consumption in hardware development [32]. In Scotch, the supported models and ScotchDSL provide enough freedom to implement popular sketching algorithms while limiting the flexibility to preserve DR4.

8 AUTOMATED TUNING

Scotch uses an auto-tuning algorithm to maximize the sketch size within the provided clock frequency constraints and resource limitations (DR2). Hence, it maximizes the accuracy of the sketch. In a nutshell, the algorithm performs an initial compilation, projects the maximum possible FPGA size within the FPGA's critical resources, and performs a binary search to find the maximum sketch size still generating functional hardware.

Before tuning, the user fixes one of the matrix dimensions while the other is subject to optimization. Row and column sketches inherently fix one dimension. Matrix sketches usually have error guarantees in the form of an interval around the true value. The number of rows m determines the probability of the estimate falling into the interval; the number of columns n determines the interval's size [9, 11]. While the algorithm can optimize either dimension, a user is likely to set the probability by fixing m based on the application and let Scotch minimize the interval by maximizing n.

As the FPGA toolchains are proprietary, we base our algorithm on the following conservative assumptions:

A1: Linear Resource Consumption. We assume ELU and BRAM consumption increases asymptotically linearly with the number of rows or columns. We use A1 to compute an upper bound for the optimization parameter. We base the assumption on the following observation: Increasing the number of rows results in a linear increase in the number of compute units and state memory components. The number of 4-dispatch, 4-collect units, and row counter bits in the State Transfer Controller grow logarithmically. Increasing the number of columns leads to a linear increase in memory banks and DFU stages. The number of bits required for offset registers and the number of LUTs for logic operating on them (e.g., comparisons) grow logarithmically.

A2: Global Optimum. We assume that there is a parameter b such that all parameters $x \leq b$ provide a functioning accelerator while all $x > b$ will lead to the optimization either failing due to timing or lack of resources. A2 justifies using a binary search. We base the assumption on the following intuition: As established in Assumption 1, resource consumption increases monotonically with the optimized parameter. Thus, FPGA resources will eventually exceed. Before this is the case, placement and routing by the toolchain, while satisfying timing, gets increasingly challenging and eventually impossible. In particular, the maximum operating frequency for the sketching unit decreases monotonically, which is the prevailing cause for timing failures.

Our auto-tuning algorithm consists of two steps:

Step 1: Initialization. The algorithm calls the RTL generator for an initial parameter r and compiles the accelerator using the vendor toolchain. Scotch estimates a parameter that exceeds 100% resource utilization and serves as a potential upper bound u. If the compilation for r has been successful, we define the initial search interval as $[r, u)$. If the compilation was not successful due to timing or resources, the interval is $[0, min(d, u))$.

Step 2: Binary Search. The algorithm performs a binary search in the interval. It repeatedly compiles the accelerator and checks whether compilation was successful. As the algorithm converges, the lower bound either contains the maximum parameter with a successful compilation or is zero if no such parameter exists.

In Scotch, we extend the basic auto-tune algorithm by following adaptations: (1) A user may choose a relative difference between upper and lower bound to speed up convergence. (2) The vendor toolchains use randomized algorithms for placement and routing. We account for their variance by trying five initial seeds for Intel Quartus Prime before considering a parameter failed due to timing. For Xilinx Vivado, we vary implementation strategies for the same effect. (3) When BRAM is depleted, vendor toolchains try to implement remaining memory banks less efficiently using ELUs. While only feasible if the fixed parameter is small, we double the estimated upper bound for row and matrix sketches to account for this edge case.

9 EVALUATION

In this section, we evaluate the RTL generator and autotune algorithm. The Scotch system, algorithm implementations, and baselines are available in our public repository. [1]

[1] https://github.com/martinkiefer/scotch

9.1 Experimental Setup

We implement two column sketches ($n = 1$), two matrix sketches, and two row sketches ($m = 1$) as shown in Table 2. We order the three sketch types by the number of updates applied to the state matrix for each input value.

Table 2: Sketching algorithms implemented

Column	AGMS [2]	MinHash (MH) [4]
Matrix	FAGMS [9]	Count-Min (CM) [11]
Row	Fast-Count (FC) [34]	HyperLogLog (HLL) [15]

We use the H3 family [27] for hashing values into arbitrary integer range in CM, FAGMS, MH, and HLL. For FC, we use an adaptation of the Polynomials-over-Primes scheme with a Mersenne Prime [27] to obtain a 4-wise independent hash function. For AGMS and FAGMS, we use the EH3 family for +1/-1 hashing [28]. We implement all algorithms for 32-bit input values. The state values for HLL are 6 bits wide; all other algorithms use 32-bit states.

Intel FPGA (A10, S10): We generate accelerators for two different Intel FPGA models: A midrange model (A10, Arria 10 GX 1150) and a high-end model (S10, Stratix 10 GX 2800). We use Intel Quartus Prime 19.3 as the vendor toolchain and Intel's Early Power Estimator to compute the power consumption.

Xilinx FPGA (XUS+, XUS): We generate accelerators for a recent Xilinx UltraScale+ FPGA (XUS+, XCVU7P) and an UltraScale FPGA (XUS, XCVU440). We use Vivado v2020.01 as the vendor toolchain.

CPU (Xeon): We run the algorithms on an Intel Xeon Silver 4214 with two sockets, each providing 24 hyper-threads. We use GCC 7.5 with OpenMP and vectorization compiler intrinsics (AVX512). We measure power consumption using powerstat 0.02.22 [19].

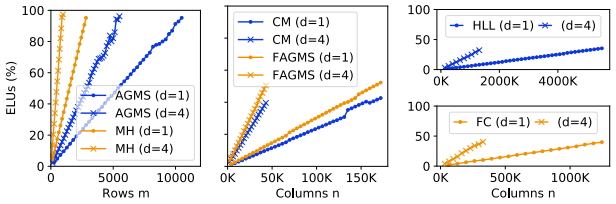
GPU (GeForce): We run the algorithms on an Nvidia GeForce RTX 2080 GPU using CUDA 10.2. We measure the power consumption using the nvidia-smi tool provided by CUDA.

FPGA accelerators use a minimal I/O template to focus on the performance of the sketching units in isolation. We use the Map-Apply model for column sketches if not stated otherwise. Our CPU and GPU baselines are hand-optimized, data-parallel, and fully utilize the architecture's parallelism. Measurements were taken on a machine running Ubuntu 18.04 for 20 iterations. We used a 2 GB uniform data set residing in main memory for the CPU and device memory for the GPU.

9.2 RTL Generator

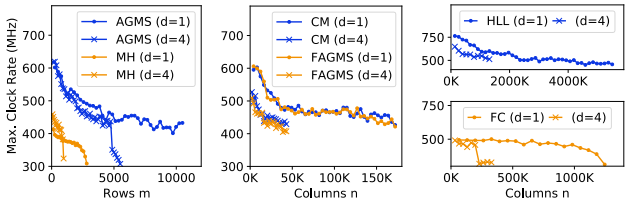
We investigate the scaling behavior of the generated accelerators in terms of resource consumption and maximum operating frequency. For the sake of readability, we restrict to matrix sketches with $m = 8$. We vary the matrix size used by the RTL generator and compile with five different compilation seeds. We report the results for A10 as it has the lowest compile times (4-20x compared to S10) due to fewer resources and, thus, allowed for a more fine-grained analysis of the parameter space. Experiments for S10 and XUS+ have shown similar results and did not provide additional insights.

9.2.1 Resource Consumption. We investigate the resource consumption of generated sketching units for varying matrix sizes and data parallelism degrees $d = 1$ and $d = 4$. In particular, this



(a) Column (b) Matrix (c) Row

Figure 9: ELU consumption for various sketching units



(a) Column (b) Matrix (c) Row

Figure 10: Max. clock frequency for various sketching units

allows us to validate that resource consumption increases linearly with the number of rows and columns (A1). Throughout all experiments, the number of BRAM blocks consumed is precisely the number of blocks assigned for the state memory. The ELU consumption for all algorithms is given in Figure 9. As ELU consumption varied under 0.01% for different seeds, we only report the maximum value. Overall, we observe that ELU consumption is approximately linear for all algorithms and both degrees of parallelism. Increasing the degree of data parallelism leads to a roughly proportional increase in resource consumption. For the column sketches shown in Figure 9a, we see that MH has an up to 6 times higher ELU consumption, which is due to a more involved map and apply function. For matrix sketches in Figure 9b, FAGMS variants show an up to 30% higher ELU consumption than CM sketches due to more complex update logic. For the row sketches in Figure 9c, we observe HLL supporting much larger summary sizes, which is due to its smaller state. Neither matrix nor row sketches exceed 55% ELU utilization as BRAM blocks are the limiting factor.

Summary: The experiment confirms that resource utilization is linear (A1). We observe that data parallelism comes at the cost of higher resource consumption.

9.2.2 Maximum Operating Frequency. Next, we investigate the impact of the state matrix size on the maximum operating frequency. As a low maximum operating frequency is the prevailing cause for failed timing, this experiment allows us to validate A2 of the auto-tuning algorithm. The maximum operating frequency varied by up to 110 MHz for different compilation seeds; therefore, we report the maximum over five runs.

Figure 10 shows the results. We observe that the maximum operating frequency decreases with a growing state matrix for all variants. However, it does not decrease strictly monotonically due to remaining variance. Almost all algorithms show a lower frequency for $d = 4$ consistent with the increased resource consumption. In Figure 10a, we see MH being the only exception to this, which we consider a toolchain artifact. We see that AGMS operates at an up to 200 MHz higher clock frequency than MH due to its simpler map and apply function. For the matrix sketches shown in Figure 10b, we see operating frequencies between 400 and 600 MHz. Both algorithms decrease in a similar L-shaped pattern. For row sketches shown in Figure 10c, we observe HLL starting at a higher frequency of up to 763 MHz due to the significantly smaller state. FC starts at 500 MHz, decreases flatly, and shows drops of more than 120 MHZ for the largest sizes, indicating that its arithmetic-based hash function has become harder to place and route.

Summary: As the experiments confirm a trend towards a decreasing maximum operating frequency, A2 of the auto-tune algorithm is justified. While the algorithm may miss the global optimum due to remaining variance, it is an efficient alternative to a prohibitively expensive exhaustive search.

9.2.3 Impact of the Map-Apply Model. Finally, we conduct experiments investigating the benefits of the Map-Apply model for column sketches incorporating data parallelism. Figure 11 compares Map-Apply to replication for column sketches and a data parallelism degree of four. In terms of ELU consumption, we see that AGMS benefits most from Map-Apply, allowing for implementations that consume only half the resources and, thus, for a more than twice as high maximum number of columns. MH shows a smaller improvement of 25% due to the more expensive update logic that reduces the positive effect of shared resources in Map-Apply. Map-Apply also shows a positive impact in terms of the maximum operating frequency. It results in an up to 80 MHz improved frequency for AGMS and an up to 60 MHz improved frequency for MH.

Summary: The Map-Apply model offers improved ELU efficiency and operating frequencies for data-parallel accelerators.

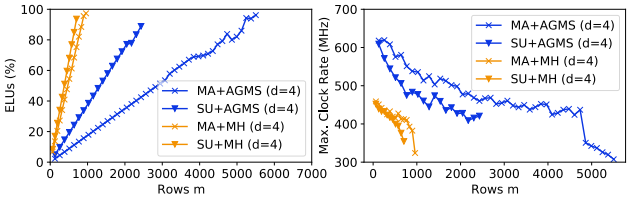
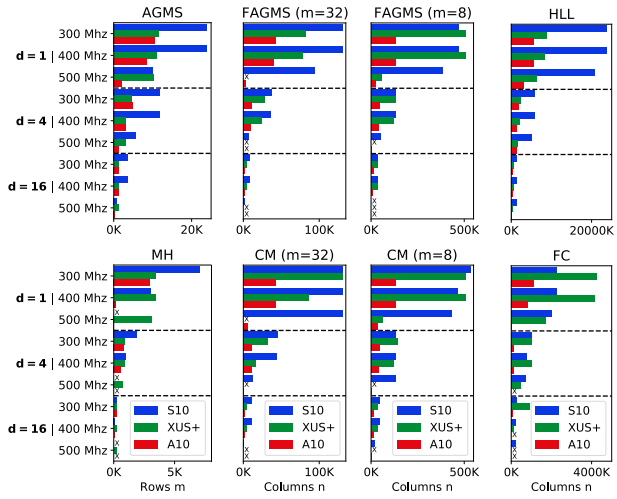


Figure 11: Comparison of Map-Apply (MA) and Select-Update with replication (SU) for data parallel accelerators

9.3 Automated Tuning

In the second set of experiments, we evaluate the performance of accelerators found by the auto-tuning algorithm in terms of summary size, throughput, and energy consumption. We generated accelerators for operating frequencies of 300, 400, and 500 MHz. Tuning starts with an initial parameter of 16 and stops at a relative difference of one percent. Tuning took between two hours and a week for A10, between four hours and six days for XUS+, and between one day and two weeks for S10.

9.3.1 Summary Size. First, we present the summary sizes found for the generated accelerators. This experiment allows us to showcase the effects of different target clock rates, degrees of parallelism, and device types for the generated sketches on the summary size.



(a) Column **(b) Matrix** **(c) Row**

Figure 12: Summary sizes for FPGA accelerators on varying target devices with varying operating frequencies

Figure 12a shows the results for column sketches. We see that all devices are capable of generating accelerators at all operating frequencies for AGMS. For MH, with its more involved select and update function, summary sizes are between three and 49 times smaller. 500 MHz accelerators are either not possible for S10 and A10 or have less than five counters, while summary sizes for XUS+ are less affected by the increased operating frequency.

Figure 12b shows the results for matrix sketches. We observe that both algorithms perform similary for the same number of rows as FAGMS and CM only differ in their update function. Decreasing the number of rows from $m = 32$ to $m = 8$ results in a roughly proportional increase of the summary size. As BRAM blocks are used, timing behavior for XUS+ changes as indicated by 500 MHz accelerators barely being possible. However, for 300 and 400 MHz, XUS+ even provides throughput competitive to S10. Figure 12c shows the results for row sketches. For HLL, we see S10 providing the largest summary sizes by up to a factor of 12. However, for FC, we see XUS+ providing up to 6.5 times larger summary sizes for $d = 1$ and $d = 4$ at 300 and 400 MHz. This observation adds to the impression that XUS+ is better at implementing large rows with a 32-bit state than the other devices.

Overall, we see that increasing the degree of data parallelism always results in a decreasing maximized parameter. The decrease is not always proportional due to the effects of the target operating frequency. Increasing the operating frequency usually leads to a smaller parameter, especially when comparing 400 and 500 MHz.

Summary: We see that the maximum summary size varies strongly depending on the algorithm, FPGA, target operating frequency, and parallelism degree. This shows the auto-tuning algorithm tailoring the summary size to the setup.

9.3.2 Throughput. Next, we investigate our accelerators' throughput compared to a state-of-the-art CPU (Xeon) and GPU (GeForce). For the sake of brevity, we report on the larger S10 and XUS+ devices operating at 400 MHz. Our accelerators' throughput is $f \cdot d \cdot 32$ due to a static processing rate f.

Figure 13a shows the results for the column sketches. We see that our FPGA accelerators outperform the Xeon and GeForce baseline in all cases. Even the more competitive GeForce baseline shows an improvement ranging between a factor of 17 and 122 for AGMS and a factor of 13 to 60 for MH. FPGAs fully leverage their potential for column sketches: We generate hardware that updates tens of thousands of counters every clock cycle. CPU and GPU implementations need several instructions for an update and can not adjust all counters simultaneously due to limited parallel compute resources.

Figure 13b shows the results for CM with 8 and 32 rows as a representative for matrix sketches. We omit FAGMS as the results barely differ. Compared to our Xeon implementations, we see that all FPGA accelerators outperform it. For the GeForce baseline, we see a more competitive throughput: For $m = 32$, S10 and XUS+ need a data parallelism degree $d = 4$ to provide a clear advantage. For $m = 8$, the throughput provided by GeForce increases by a factor of four as the total amount of computations per input value has decreased by the same factor. However, the device can not achieve this throughput (entire bar) in practice, as the data transfer bandwidth (filled bar) limits the processing throughput. This effect is well-known as the PCIe bottleneck. With a data parallelism degree of $d = 16$, the FPGA accelerators outperform GeForce by a factor of up to 2.6 considering the PCIe bottleneck and 10 to 40% when input data resides in device memory. FPGAs support various interconnects allowing them to overcome transfer bottlenecks [32].

Figure 13c shows the results for row sketches. As before, FPGA accelerators with $d = 16$ are sufficient to outperform GeForce. When disregarding the PCIe bottleneck, we see two cases: For $n < 700k$, GeForce provides throughput up to an order of magnitude above the transfer bandwidth as only one update per data item is performed. For higher n, throughput decreases drastically due to cache misses, highlighting the impact of data dependencies.

Summary: The experiments show that automatically tuned FPGA accelerators can outperform software implementations on parallel architectures in many cases. For matrix and row sketches, data parallelism is essential to outperform a GPU.

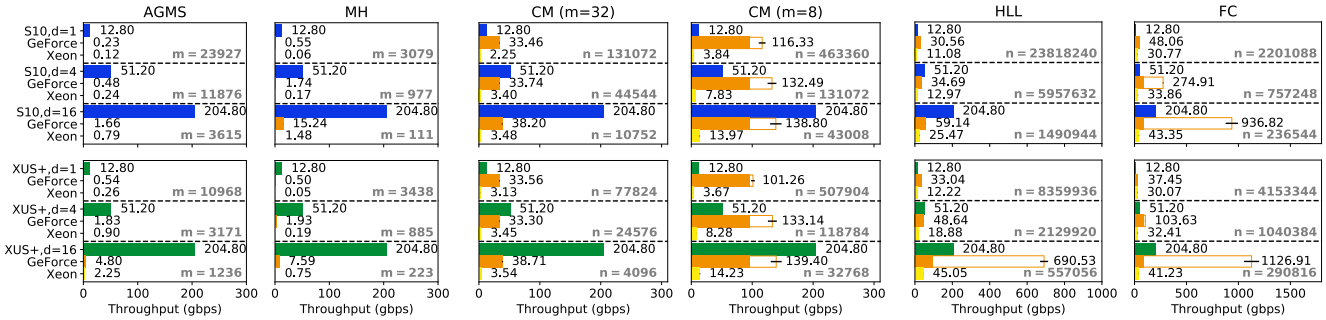
9.3.3 Power Consumption. Next, we evaluate the power consumption of our sketching accelerators. We generate accelerators on Stratix 10 for AGMS, CM with $m = 8$, and HLL as representatives for column, matrix, and row sketches for 100 Gbit/s throughput (390.625 MHz, $d = 8$). We compare the power consumption to our CPU and GPU baselines at peak throughput. Measurements are obtained using software and do not include periphery.

Table 3 shows the results. We find FPGA accelerators consuming between 2.5 and 5.6 times less energy than Xeon and GeForce.

Table 3: Power consumption in Watt for 100 Gbit/s accelerators (S10, 390.625 MHz, $d = 8$) compared to baselines

Sketch	S10	Xeon	GeForce	Size
AGMS	33.05	152.79	184.32	n=1, m=6288
CM	69.85	176.62	213.99	n=84992, m=8
HLL	50.48	145.91	124.06	n=2981888, m=1

Summary: The experiment shows that FPGA accelerators consume significantly less power than the CPU and GPU baselines.



(a) Column **(b) Matrix** **(c) Row**

Figure 13: Throughput for FPGA accelerators compared to Xeon and GeForce baselines

9.4 Comparison to Hand-Written RTL

Finally, we compare Scotch to a state-of-the-art hand-written sketching implementation. Tong and Prasanna implemented CM sketching on XUS [35]. Their sketching approach is comparable as they guarantee static data rates and utilize a pipelined memory architecture with a DFU and $d = 1$. We compare their reported operating frequency, throughput, and summary size to an Scotch accelerator generated with the same FPGA and operating frequency. Table 4 shows Scotch found an implementation with twice as many columns.

Summary: Scotch competes with a manual implementation.

Table 4: Scotch compared to [35] for CM (XUS, $m = 5, d = 1$)

Implementation	Operating Freq.	Throughput	Columns n
Hand-Written [35]	497 MHz	15.9 gbps	2^{16}
Scotch	503 MHz	16 gbps	2^{17}

10 RELATED WORK

Accelerating the construction of sketch summaries and their applications on FPGAs has been proposed in previous work. Our approach is related to the work of Tong and Prasanna, who used FPGAs for online heavy hitter and change detection in high throughput networks [35]. They implement Count-Min sketching with guaranteed data rates using pipelined memory and a data forwarding unit. As shown in Section 9.4, our accelerators can operate at the same frequency and throughput for a data parallelism degree of one. However, we can provide even higher throughput by exploiting data parallelism. Other streaming algorithms implemented on FPGAs are Space Saving [31], Exponential CM [8], CM-CU [29], MinHash [30], and Bloom Filters [7].

High-level synthesis tools such as OpenCL [6, 12, 39] or VivadoHLS [14] allow for the generation of accelerators from C-like programming languages. While these frameworks are general purpose and have been successfully applied in database acceleration [37, 38], Scotch exploits knowledge about the memory access pattern and semantics defined by the select-update and map-apply models to provide fully pipelined sketching RTL; the tuning process is automatic, given a matrix size and user defined functions. However, high-level synthesis tools can reduce development effort and expertise required to develop I/O templates for Scotch.

Previous research has suggested automated tuning of toolchain and user parameters towards an objective function [5, 20]. It treats RTL and optimization parameters as a black box and, thus, requires general models and sufficient training. Scotch's automated tuning algorithm uses intuitive assumptions based on domain knowledge to optimize for the summary size using a practical approach that is logarithmic in the search space.

11 CONCLUSION

In this paper, we introduced Scotch, a novel system for generating optimized sketching accelerators on FPGAs. It provides a full system stack covering all aspects, from sketch specification over code generation to automated tuning.

We evaluated Scotch for six sketching algorithms and three different FPGAs. We showed that Scotch tailors the summary size to the desired throughput and FPGA. We highlighted the inherent trade-off between throughput and summary size controlled via the operating frequency and degree of parallelism. We found that the accelerators can satisfy interconnects with a bandwidth of 100 Gbit/s and more. Scotch accelerators outperform CPU baselines by a factor of up to 300 in terms of throughput and by a factor of up to 4.6 in terms of energy consumption. Compared to a GPU baseline, FPGA accelerator throughput ranges from competitive to an improvement of a factor of 120 while consuming up to 5.6 times less energy. Furthermore, we found that Scotch accelerators compete with a manual FPGA implementation.

Overall, this paper shows that Scotch produces highly efficient FPGA sketching accelerators without manual RTL programming and tuning. Thus, Scotch substantially lowers the entry bar for FPGA accelerated sketching by replacing an FPGA expert with code generation and automated tuning.

ACKNOWLEDGMENTS

This work has received funding by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A and 01IS18037A) and Software Campus (01IS17052).

REFERENCES

[1] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable Summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 23–34.

[2] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking Join and Self-Join Sizes in Limited Storage. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 10–20.

[3] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] A. Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of SEQUENCES 1997*. 21–29.

[5] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic. 2017. Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp. In *2017 International Conference on ReConFigurable Computing and FPGAs*. 1–6.

[6] D. Chen and D. Singh. 2012. Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering. In *22nd International Conference on Field Programmable Logic and Applications*. 5–12.

[7] J. M. Cho and K. Choi. 2014. An FPGA implementation of high-throughput key-value store using Bloom filter. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. 1–4.

[8] G. Chrysos, O. Papapetrou, D. Pnevmatikatos, A. Dollas, and M. Garofalakis. 2019. Data Stream Statistics Over Sliding Windows: How to Summarize 150 Million Updates Per Second on a Single Node. In *29th International Conference on Field Programmable Logic and Applications*. 278–285.

[9] Graham Cormode and Minos Garofalakis. 2005. Sketching Streams through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 13–24.

[10] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1–3 (2012), 1–294.

[11] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to High-Performance Hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications*. 531–534.

[13] Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. 2003. An Approximate L1-Difference Algorithm for Massive Data Streams. *SIAM J. Comput.* 32, 1 (2003), 131–151.

[14] Tom Feist. 2012. Vivado design suite. *Xilinx White Paper* 5 (2012), 30.

[15] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the 2007 Conference on Analysis of Algorithms*. 127–146.

[16] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31, 2 (1985), 182–209.

[17] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1647–1660.

[18] Amit Goyal, Hal Daumé, and Graham Cormode. 2012. Sketch Algorithms for Estimating Point Queries in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 1093–1103.

[19] Colian Ian King. 2020. *Powerstat.* Retrieved November 10, 2020 from https://github.com/ColinIanKing/powerstat

[20] M. Kurek, M. P. Deisenroth, W. Luk, and T. Todman. 2016. Knowledge Transfer in Automatic Optimisation of Reconfigurable Designs. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*. 84–87.

[21] Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. 2011. Hashing Algorithms for Large-Scale Learning. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*. 2672–2680.

[22] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[23] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory*. 398–412.

[24] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Data Processing on FPGAs. *Proc. VLDB Endow.* 2, 1 (2009), 910–921.

[25] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *Proceedings of the VLDB Endowment* 2, 1 (2009), 229–240.

[26] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-Based Querying of Distributed Sliding-Window Data Streams. *Proceedings of the VLDB Endowment* 5, 10 (2012), 992–1003.

[27] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. 1997. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput.* 46, 12 (1997), 1378–1381.

[28] Florin Rusu and Alin Dobra. 2007. Pseudo-random number generation for sketch-based estimations. *ACM Transactions on Database Systems* 32, 2, Article 11 (2007).

[29] A. Saavedra, C. Hernandez, and M. Figueroa. 2018. Heavy-Hitter Detection Using a Hardware Sketch with the Countmin-CU Algorithm. In *2018 21st Euromicro Conference on Digital System Design*. 38–45.

[30] Javier Soto, Thomas Krohmer, Cecilia Hernandez, and Miguel Figueroa. 2019. Hardware Acceleration of k-Mer Clustering using Locality-Sensitive Hashing. In *22nd Euromicro Conference on Digital System Design*. 659–662.

[31] J. Teubner, R. Mueller, and G. Alonso. 2010. FPGA acceleration for the frequent item problem. In *IEEE 26th International Conference on Data Engineering*. 669–680.

[32] Jens Teubner and Louis Woods. 2013. Data processing on FPGAs. *Synthesis Lectures on Data Management* 5, 2 (2013), 1–118.

[33] Jens Teubner, Louis Woods, and Chongling Nie. 2013. XLynx-An FPGA-Based XML Filter for Hybrid XQuery Processing. *ACM Transactions on Database Systems* 38, 4, Article 23 (2013).

[34] Mikkel Thorup and Yin Zhang. 2004. Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 615–624.

[35] D. Tong and V. K. Prasanna. 2018. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 929–942.

[36] David Vengerov, Andre Menck, Mohamed Zait, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1530–1541.

[37] Zeke Wang, Bingsheng He, and Wei Zhang. 2015. A study of data partitioning on OpenCL-based FPGAs. In *25th International Conference on Field Programmable Logic and Applications*. IEEE, 1–8.

[38] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. 2016. Relational query processing on OpenCL-based FPGAs. In *26th International Conference on Field Programmable Logic and Applications*. 1–10.

[39] Loring Wirbel. 2014. Xilinx SDAccel: a unified development environment for tomorrow's data center. *The Linley Group Inc* (2014).

[40] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[41] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH ensemble: Internet-scale domain search. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1185–1196.