TU Delft

MSc ES & CE

Thesis

Automated FPGA Hardware Synthesis for High-Throughput Big Data Filtering and Transformation

An SQL query transpiler targeting Vivado HLS C++ tools for high-level stream transformation and filtering on FPGAs using Apache Arrow.

Erwin de Haan
e.r.de.haan@student.tudelft.nl

August 20, 2019

Automated FPGA Hardware Synthesis for High-Throughput Big Data Filtering and Transformation

An SQL query transpiler targeting Vivado HLS C++ tools for high-level stream transformation and filtering on FPGAs using Apache Arrow.

A Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

and

Embedded Systems

by Erwin Raynald de Haan

To be defended on Tuesday the 27th of August

Version: 1.0.0

Thesis no. Q&CE-CE-MS-2019-12
by E.R. de Haan (4222814)

Faculty of EEMCS
Delft University of Technology
Mekelweg 4
Delft, The Netherlands

Abstract

Despite its advantages in performance and control, hardware design is mainly bottlenecked by high design complexity and long development time. This thesis explores the use of domain specific languages for high-level synthesis (HLS) of hardware data filters and transformations.

The main goal of this thesis' prototype is automating the transpiling of SQL to HLS C++ to generate hardware for filtering and data streams using CAPI on POWER systems. This work uses the Fletcher framework to automate the handling of data movement between memory and the field-programmable gate array (FPGA). The use of HLS technologies can greatly reduce the development time of FPGAs compared to manual FPGA development workflows. Deploying FPGAs in fast changing data processing pipelines, can be very complicated or limit the use of the FPGA hardware. This work investigates if HLScan be used for these kinds of applications to reduce total development time while still maintaining performance. Additionally, the use of the Fletcher framework further reduces required developer time. The proof-of-concept shows that it is possible to efficiently use HLS for data filtering and transformations. And that without a significant effort from the designer, usable designs and filters can be generated. For example some of the simpler kernels can reach upwards of 1 GB/s while using less than 1 % of a Xilinx Kintex UltraScale XCKU060 FPGA. By using multiple instances of these kernels the design can saturate the system bandwidth. Though this approach is not without issue, it does lend itself to extending the tool and some extra development effort to improve the current proof-of-concept.

The project code is released under Apache 2.0 license on GitHub at:
https://github.com/EraYaN/FletcherFiltering.

Preface

This thesis is the result of a nine month effort to see if HLS is suitable to integrate with Fletcher, a framework that the Accelerated Big Data Systems group at the Delft University of Technology has been working on; and to see if HLS is a good solution for the task of connecting multiple different kernels. The desire for a query compiler or a easier way for data scientists and analysts to interface with the framework and FPGAs, resulted in the effort this thesis describes. When used in a Just-in-Time situation this can help keep the data on the FPGA for as long as possible, since then filter operations between different compute kernels do not need to move the data off the chip.

This thesis describes the design, implementation and testing of this compiler, and answers the question if HLS is ready and mature enough yet, to in a reasonable time-frame develop such an application, without sacrificing too much of the performance of the resulting output hardware.

These past nine months have been a great learning experience for me, from internal compiler design philosophies to the nitty-gritty details of the hardware design and deployment tools.

I would like to thank my friends and colleagues in the Accelerated Big Data Systems group for their tremendous help and assistance, Peter Hofstee for his assistance in getting the testing infrastructure ready and working, my copyreaders for helping find my mistakes and shaping the final text, and my supervisor Zaid Al-Ars for his feedback during the process of development and writing.

— Erwin de Haan, 20 August 2019

Contents

1 | Introduction

1.1 Context

In the race to more and bigger data systems, performance is becoming more and more the focus. This increased focus paves the way for accelerating big-data tasks with accelerators. FPGAs offer a great opportunity for big data systems due to their ability to provide very wide and deep data pipelines and the resulting very high throughput potential. Using FPGAs in an ever-changing environment is a complex problem though. Hardware design is much more expensive than software design and development cycle times are much longer. This may lead to high or even prohibitive costs for any application with evolving data processing needs.

To mitigate this problem, FPGAs vendors have been hard at work making HLS tools to lower the bar of entry for FPGA development. These allow people to use higher level languages such as C++ to describe hardware processes. The downside is that it still requires some hardware knowledge to effectively make use of these tools, although resulting performance has been getting better [23].

HLS faces some real challenges in reaching the same or similar performance as hand-written hardware description language (HDL) designs without a lot of care and coaching from the developer's side. Different tools can vary a lot between different benchmarks and the spread in performance is quite significant [16]. For some applications HLS can reach the manually optimized register-transfer level (RTL) descriptions' performance as shown by Homsirikamol and Gaj [9]. This does, however, require a lot of attention from the developer. HLS code needs extra attention for things like properly pipelining and unrolling loops and removing control flow. These optimizations help the HLS compiler generate more performant hardware designs. However, for most existing C++ software code many of these tools will generate highly inefficient designs. This means that, even though HLS will save developer time because of the amount of required code is much smaller, one is still writing code quite specifically for and informed by hardware architectures. This is still not ideal.

For many application domains, HLS can fill the role of the orchestrating language or otherwise for simpler glue kernels, between bigger and more complex HDL IP cores. One might think of it as the Python of the hardware world, especially with the addition of blackbox C functions, to be replaced by custom HDL designs, to the newest version of for example Vivado HLS.

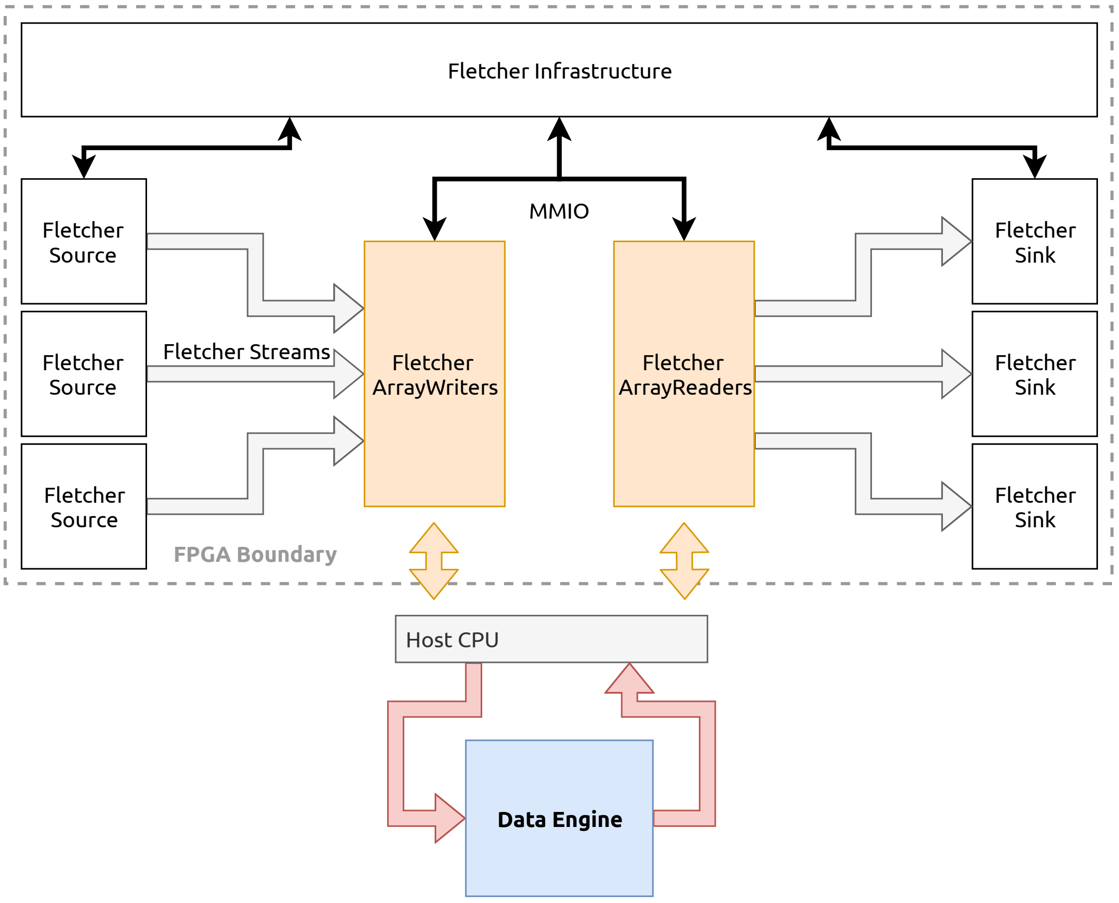


Figure 1.1: Filtering on the CPU adds a full roundtrip to host memory. All existing CPU tools for data filtering and transformation are available.

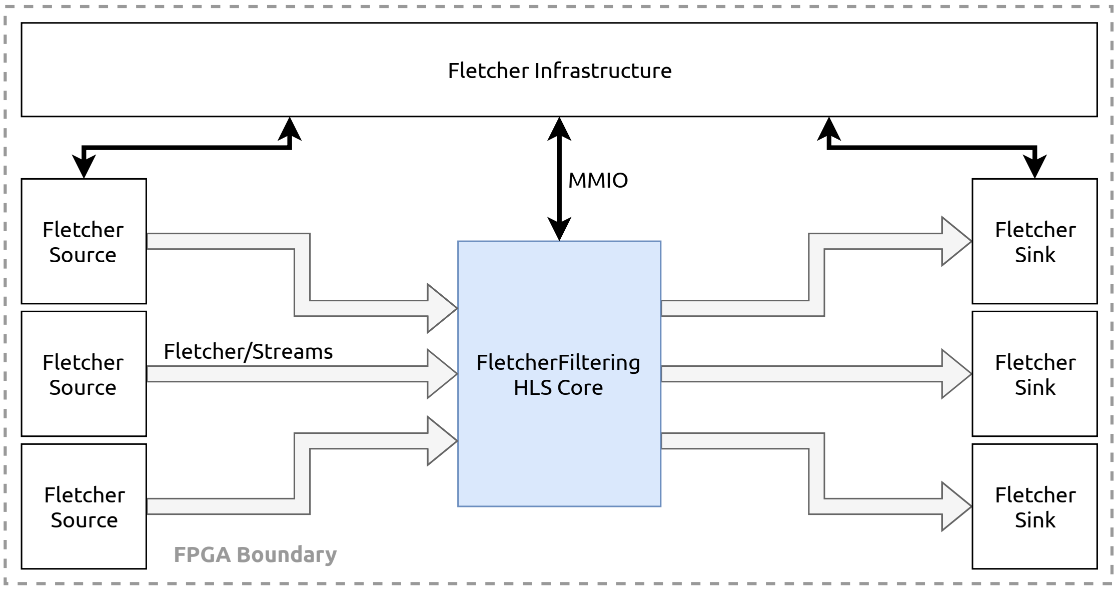


Figure 1.2: Filtering on the FPGA without the round trip to the CPU. More limited filters, but most common cases are be covered.

1.2 Problem definition & research questions

The main problem is illustrated in figure 1.1. On the left, there are sources of data, be it other kernels or for example an `ArrayReader` that reads directly from memory. On the right, there are data destinations, which consume the data. Consider a scenario where there needs to be a filter operation between these two sides. For example an outlier filter, or a date range filter. The filter runs on the host CPU, meaning all the data needs to go to host memory for the CPU to be able to process it. For big data applications, one never wants the data to touch main memory if it can be helped. So this is a problem, figure 1.2 illustrates the proposed solution. Here in the middle is a core generated by HLS that does the filtering, so that the data never has to leave the FPGA. The core can filter or transform the data or both, all using the only on-chip resources. Of course, the type of filters will be more limited than on the host CPU, but for many cases, this should provide much better performance.

This scenario reflects a use case where this setup would be used as a just-in-time kernel generator in a big data framework. Mostly when there is a simple `SELECT` or `WHERE` operation between two bigger fixed kernels. Those fixed kernels would be shipped with the framework, not unlike how GPU kernels are shipped with frameworks today. This setup would allow for these little filters and transformations to the automatically generated thus preventing a move of the data back over to the host memory, and instead it can keep all the data on-chip. This project could be used to make longer pipelines out of multiple of these kernels.

Considering all of this, a couple of research questions arise:

How much of a performance bottleneck would HLS tools introduce into the generated hardware?

Can HLS C++ be used as an orchestration language for hardware?

Can one automate most of this HLS code writing for filtering and simple transformations?

Does this automation result in saved developer hours?

1.3 Outline

This thesis explores the possibility of using a domain specific language to generate such HLS code, to reduce the developer time cost even further for making these smaller filter and stream transformation kernels. As an example the SQL query language is used to describe filters and stream data transformations. This results in a SQL to HLS C++ transpiler, to be used for example as a just-in-time component in a larger data processing system or as some glue logic to connect to other computational cores.

This thesis consists of several parts, first a look at the background and related work in chapter 2. Then a look into the design goals and implementation in chapters 3 and 4. The specification of all the performed tests is described chapter 5. An overview of the test cases' results is discussed in chapter 6 and finally the conclusions and recommendations in chapter 7.

2 | Background

2.1 FPGAs & SQL

In the last couple of years, FPGAs have slowly started to make their way into the high-performance computing (HPC) and the Big Data world. Their incredible potential for complex parallel processing makes them very appealing to many in this space. A large amount of research effort has gone into finding uses for FPGAs in the HPC world, with a lot of very promising results [8, 13]. Sadly not all algorithms are very suitable for FPGAs, some are much more efficient on GPUs for example, like kernels that are very heavy on floating point compute [10]. Still, other applications can benefit from great speed-ups, lower latency or power savings [17].

SQL as a language has long been used for filtering and transforming data. It has a very extensive syntax and allows many constructs outside just filtering and transformation. This makes it harder to fully master, but plain `SELECT` statements are very easy to pick up. Many in the data analysis field are familiar with the language, making it a good pick for the interface language of this project. The project will support a subset of SQL, mainly plain `SELECT` statements with `WHERE` clauses, as described in section 3.3.

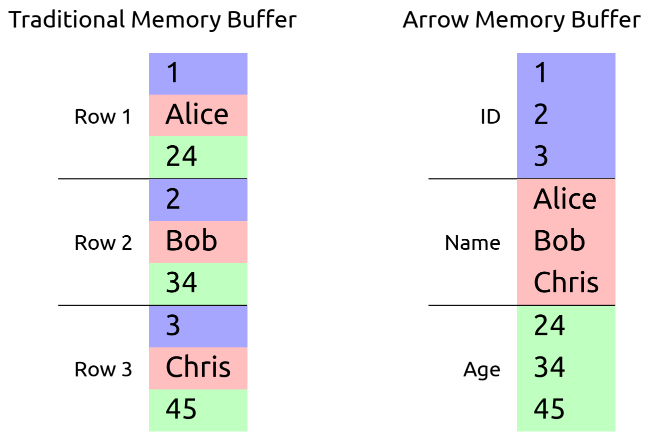
2.2 Arrow & Fletcher

More and more large scale applications are constrained in performance by data movement. Some often used technologies and platforms use different in-memory layouts, making this even worse, since this will cause a (de)serialization overhead every time data passes between these different entities. To help with the data movement and serialization overhead, Apache Arrow tries to define a common memory model [1]. This memory model can greatly reduce the amount of data copying and transformation that needs to be done to have different programming environments cooperate. The Fletcher project brings this technology to FPGAs [7], this enables FPGAs to directly use the host memory buffers without any data serialization. The Fletcher project includes tools to generate all the required hardware to easily read and write to the Arrow buffers either on the device memory or when using Coherent Accelerator Processor Interface (CAPI) the host memory.

Apache Arrow's standard describes a columnar format. The focusses of the project are interoperability, flexibility and performance, while being very flexible for different data types and structures [2]. A simple example is shown in table 2.1. This means if there needs to be any processing on one column, there is great data locality, and thus good performance. An example of one such operation is a filter operation in on one data field.

Table 2.1: Arrow memory buffers versus traditional memory buffers.

ID	Name	Age
1	Alice	24
2	Bob	34
3	Chris	45



2.3 Related work

Previous work on query compilers for FPGAs [14, 15] have mostly been for self contained systems without the advantage of zero-serialization integration with other cores and host-side CPU based software. These provide a great amount of functionality for processing data, but for being used as a tool in a diverse big data analysis software stack without (de)serialization overhead most of the existing work does not provide the required flexibility.

Previous work on accelerated databases has also included work that uses FPGAs as their main accelerator [19]. This work takes a hybrid approach to processing fullfeatured SQL queries: parts of the system will run on the CPU and select operators can be run on the FPGA. This setup requires a lot of bandwidth between the host processor and the accelerator. This can limit its applicability if the data needs to be shifted between compute devices often.

In the software space, there have been a number of works that have resulted in SQL to C compilers [22]. This work shows that HLS-like technique has a lot of potential also for pure software applications. The authors note that for great performance an efficient set of data structures is required. This is not fully reconcilable with the desire to share data structures in a standardized fashion amongst applications, mostly since those data structures are somewhat platform dependent: different compute environments have very different needs.

3 | Design

To test the main questions posed in this thesis, a proof of concept compiler needs to be created. The original philosophy was to use as many ready made components as possible to get something up and running quickly and try to saturate the system bandwidth with any one generated kernel or multiple of them.

3.1 Architecture

Almost every compiler consists of a front end, a host of transformations and a back end for code generation. Figure 3.1 shows the different parts of the architecture. In this case the back end needs to generate C++ code that Vivado HLS can compile to an RTL representation. While the front end needs to parse a subset of SQL (as described in section 3.3) to generate a representation that the transformations can work on. This project focusses mostly on the compiler middleware that does the transformations, so that the front end and back end can successfully be connected to produce useful output.

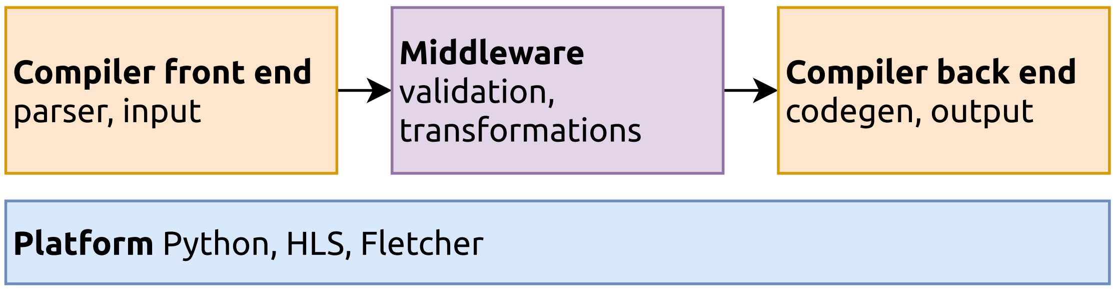


Figure 3.1: Design architecture, showing the front end, middleware and back end of the compiler.

3.2 Requirements

The compiler front end should parse a subset of SQL to an internal abstract syntax tree (AST) representation.
SQL is chosen for its ubiquity and flexibility. And ASTs are required for easy transformations in the compiler.

The compiler back end should generate correct C++ code that is understood by the HLS tool of choice, in this case Vivado HLS.

The generated HLS IP should have a signal layout that is compatible with Fletcher streams.

The one or more of the generated kernels should be able to saturate the system bandwidth.

The code should be licensed under a Apache-2.0 [3] and MPL-2.0 [12] compatible license.
This is because of the used open source dependencies.

3.3 Supported subset of SQL

The design will support a subset of SQL, because it does not need any of the control queries or things like `INSERT`, `UPDATE` and `DELETE`, because its target are stream transformations and filters. For the class of `SELECT` queries, the design does not aim to support things like `UNION` or `JOIN` for the same reason. `GROUP BY`, `ORDER BY` and `LIMIT` are also not supported. So to define the filters and transformations only `SELECT` queries with `WHERE` clauses are supported. As for the SQL functions, the function `CONCAT` is the only supported function, mostly as an example for further function implementations. Other than that all operators listed in table 3.1 are supported for booleans and numerical formats for use in the `SELECT` clause and in the `WHERE` clause.

Table 3.1: The supported operators and their applicable types.

Operator	Symbol	Types
MulOperator	*	numbers
DivOperator	/	numbers
AddOperator	+	numbers
SubOperator	-	numbers
BitAndOperator	&	integers
BitOrOperator	|	integers
BitXorOperator	^	integers
AndOperator	and	booleans
OrOperator	or	booleans
USubOperator	-	number
UAddOperator	+	number
NEqOperator	!=, <>	numbers, booleans
GtOperator	>	numbers
LtOperator	<	numbers
GtEOperator	>=	numbers
LtEOperator	<=	numbers
EqOperator	=, ==	numbers, booleans
IsOperator	is	numbers, booleans

3.4 Exploration

During the exploration phase, one of the main wants was that the kernel would be able to wrap a regular expression to VHDL generator, that was made by someone in the Accelerated Big Data Systems group. Since regular expression matching is an often used operation in text processing, this seemed like a great addition. Unfortunately the required C function replacement by RTL modules was not yet available in the then available version of Vivado HLS, since this project was started version 2019.1 was released in May which adds this functionality.

Since writing a custom SQL parser seemed quite a pointless exercise, the options that are compatible with Python are compared in table 3.2. The best option for this project is `moz-sql-parser` [11]. One of the only suitable SQL parsers out there that outputs an actual tree structure of the query, although it outputs plain dictionaries and lists, not really something that lends itself for transformations. Preferably it would be possible to use the `NodeTransformer` class from the Python standard library. Thankfully the parser is based on the very popular `pyparsing` library, making it relatively easy to change the form of the output.

Table 3.2: Comparison of alternatives for SQL parsers.

Library	**License**	**Parse Output**	**Extendability**	**Software support**
python-sqlparse	++ BSD	- Tokens	- custom regex parser	+ Python 2.7-3.3
moz-sql-parser	++ MPL-2.0	- Python dicts	++ `Pyparsing` based	++ Python 2.7-3.6
General SQL Parser	-- Commercial	- Tokens	-- Commercial	-- Python 2 only
python-sqlparser	+ LGPL	-- Internal	- In external project	+ Python 3 only

After trying to generate the C++ with some simple templating, it quickly became apparent that it would not give the project the required flexibility. In table 3.3 some of the options are compared. Since SQL allows arbitrary functions to be specified both in the `SELECT` and `WHERE` clauses the project needs a lot of flexibility in it's unparser. And using an AST or other proper intermediate representation (IR) is a much more natural fit. So in the end for the back end, the `transpyle` [4] project was picked since it provides a mostly working C++ unparser, a program that takes an AST and generates the code that would have originally produced that AST. It works on Python 3 ASTs as the input.

Table 3.3: Comparison of alternatives for C++ unparsers.

Library	License	Unparse Input	Extendability	Output quality	Flexibility
`transpyle`	++ Apache-2.0	++ Python 3 AST	++ Pure Python	++ Readable C++	++ Annotated AST based
LLVM C++	++ BSD-like	-- LLVM IR	- Complex C++	- Compilable C++	+ LLVM IR
Custom templates	++ Own code	+ Python objects	- Template engine	++ Readable C++	-- Template engine

Both of these projects, `moz-sql-parser` and `transpyle`, are fully open source and are released under respectively the MPL-2.0 [12] and the Apache-2.0 [3] licenses, both of which are compatible with this project's intended Apache-2.0 license.

Finally to connect the final HLS kernel to the generated Fletcher infrastructure, there will need to be an extra wrapper that manages the control signals and tells the Fletcher readers and writers what buffers to read or write to. It would be preferable if this wrapper is also generated automatically. During the initial phase of this project it seemed that the best idea was to extend Cerata. Cerata is part of the generation core of Fletcher, it's a library for high-level hardware design; it transforms graphs of entities into for example VHDL. Due to the complexity of this, later a templating engine in Python (Jinja2) was used for the wrapper generation.

4 | Implementation

When the compiler is used it requires an SQL query, written in a subset of SQL (as described in section 3.3), and an input and output Arrow schema both in a flatbuffer file. These schemas are the same as the ones that are used with the code generator for Fletcher, `fletchgen`. The final implementation is very extendable, if extra transformations are required they can be added to the stack fairly easily. In figure 4.1 the stack of transformations are shown on the right. On the left you have the front end, a fork of `moz-sql-parser`, more on this in section 4.1. You have the back end, a fork of `transpyle`, more on this in section 4.3. In the middle on the main block is the compiler middleware, this validates the input and passes all the required data structures to the transformations, the transformations are essentially part of the middleware. All these transformations are detailed in section 4.2. In the C++ block, the different parts of the project that are written in the C++ language are displayed, these are all support parts and are not directly used by the compiler itself. The Fletcher HLS API for example is used by the generated code and is described in section 4.4. And at the bottom of the figure, all the dependencies are shown, the ones in blue (darker on grayscale) were modified for this project. Finally in the top right is the settings module, this is meant for users to change values themselves to change the precise behaviour of the compiler and the resulting kernels.

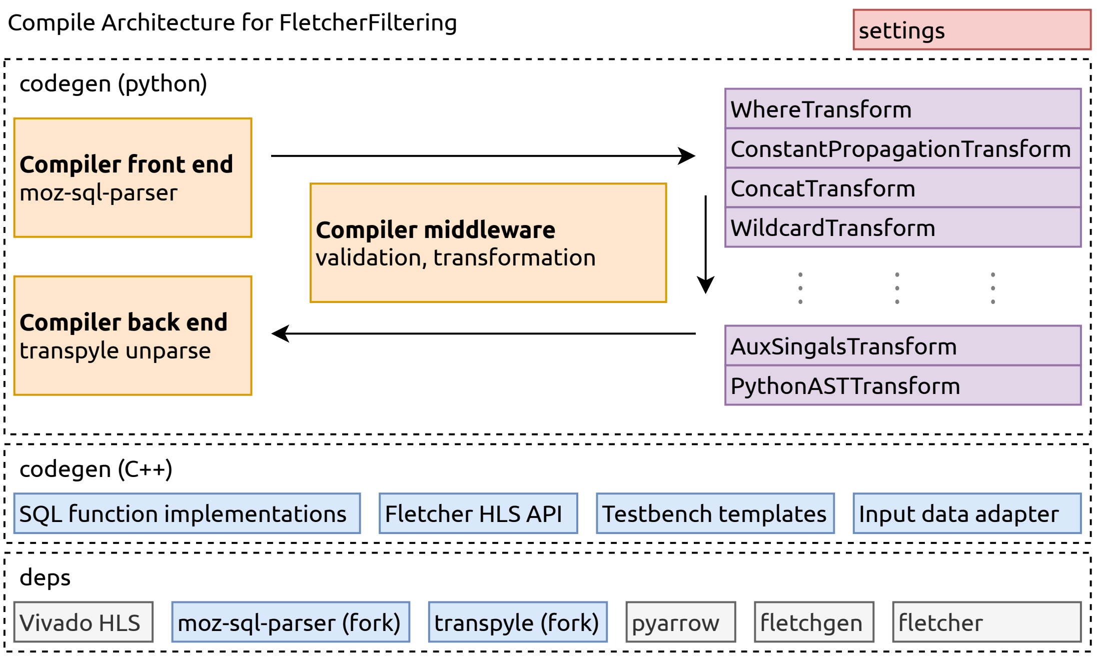


Figure 4.1: Compiler architecture, showing the front end, middleware and back end of the compiler and all the support systems, tools and libraries. This is an extended version of figure 3.1.

4.1 Front end

The already existing SQL parser, `moz-sql-parser` maintained by Mozilla written in Python, provides a good starting point, as described in section 3.4. This parser is based on `pyparsing` and is relatively easy to extend. This parser originally output object structures akin to JSON. For the first simple version this worked, but this makes transformations needlessly complicated. So as the intermediary was going to be Python 3 ASTs (sections 3.4 and 4.3), custom classes based on the `AST` Python class were added to the project and the parser adjusted to output a full AST. This AST can be properly transformed using the build in `NodeTransformer` interface, more on that later. The front end then does some sanity checking on the input and output schemas to make sure the number of columns and column names in the query match up, and after that it hands off control to the first set of transformations.

4.2 Transformations

The transformations in the middle of the compiler stack enable the correct translation of the initial custom AST into the final Python AST.

4.2.1 WhereTransform

This adds a `WHERE TRUE` clause to every query that has no `WHERE` clause. This transformation is one of the transformations that normalizes the parsed AST for transformations later in the chain.

4.2.2 ConstantPropagationTransform

This transformation does some basic constant propagation. Not necessarily to improve performance, since the target compiler will do a much better job of that. But mostly to improve the readability of the generated C++ code. This also removes adjacent variables in binary operations and literals. For example:

$$A \wedge A \mapsto true$$
$$A \vee A \mapsto A$$

This transformation runs continually until the input matches the output, this way multiple layers of constants can be propagated.

4.2.3 ConcatTransform

This is a helper transformation to transform calls to the `CONCAT` SQL function. This is mainly because there are some extra requirements because the same function call in C++. The function in SQL can have many many arguments that are all concatenated, while it's C++ counterpart only accepts a fixed number of arguments. It also adds the correct `IntermediaryReferences` for any variable that has a length, like strings. It also converts all floating point and integer types into strings and adds a length constant after them in the argument list. In the final transform the function-specific code generator will take this and generate the final Python AST for this function call.

4.2.4 WildcardTransform

This replaces any `Wildcard()` column AST node with all available input columns from the input schema. This is another transformation that normalizes the incoming AST for the last transformation, so it can be simpler in design.

4.2.5 AuxSignalsTransform

This transforms a data expression in the same expression but with any of the auxiliary signals. For example for two nullable variables A and B that are summed:

$$A.data + B.data \mapsto A.valid \land B.valid$$

The valid signals for the $+$ operator will go through an `AND` gate, this is to say that if one or both values are a NULL value the result with be NULL as well. The transform is mostly used by the PythonASTTransform if the valid signal is required separately, for example in a function call. Normally the defined fletcher data types have most operators correctly overloaded, meaning that the auxiliary signals are automatically handled behind the scenes.

4.2.6 PythonASTTransform

This in the final internal transformation of the AST. This transform builds the Python AST understood by the `transpyle` back end (section 4.3) to generate the final C++ output. This transform also uses some of the other helper transforms directly on parts of the AST, like the AuxSignalsTransform (section 4.2.5) to help it pass auxiliary Fletcher stream signals through the same operators as the main data signals.

This part of the compiler is the part that handles all the stream interaction as well, it generates the Python AST code to grab all the values out of the streams and write them to the output when appropriate. So most of the data handling that is not directly influenced by the actual query that is used is also generated here, most of the control and management code is as well.

This is also the place where the final function AST code is generated, every function call that is supported in the used subset of SQL has it's own generator function. For example the `CONCAT` function can have an unlimited number of arguments and this does not map very well to C++ and hardware, so it is split into consecutive function calls by a previous transformation, building up the final output buffer one string at a time.

The complete generated AST consists of three parts. The input and data ingestion part, the data processing and conditional evaluation part, and finally the output and stream closing part. The test functions and data structures used for the hardware and software co-simulation is also generated in this transformation by calling it multiple times with different parameters.

4.3 Back end

To make the code generation more versatile the prototype needs a suitable IR. The `transpyle` project [5, 6] uses Python 3 ASTs for it's IR and supports C++ code generation. This C++ generation needed a bit of extending, but this greatly improved the extensibility of the prototype. The final back end generates Python ASTs from the internal ASTs that come from the front end (section 4.1). These Python ASTs are passed into a slightly modified unparser from the `transpyle` framework for C++.

4.4 Fletcher HLS API

To help HLS kernels integrate with Fletcher streams, there needs to be an easy way to wrap all Arrow types in C++. There are structures for every primitive Arrow type to help users and the generated code use and move data, listed in table A.2. The aim of these types was to make them behave just like regular c-type variables, so all operators in all possible mutations are overloaded. For the nullable types, it approximately follows the rules of MySQL operators for NULL values, for example for arithmetic the valid signal passes through an AND gate as shown in section 4.2.5. In the hardware the types generate a concatenated bus (with data, last and dvalid) plus when using the `ap_fifo` interface, also a ready and valid signal, that are very similar to the Fletcher ones. Originally AXI-Stream or `ap_hs` interfaces seemed better options, but both have issues. AXI-Stream has some magic string matching going on on the names of the structure members to bind them to special AXI signals, and these are undocumented. While `ap_hs` breaks all co-simulation functionality in the Vivado suite.

4.5 Test infrastructure

The design must be tested in multiple stages. Some tests take much longer then others, so to keep development speed high early development was done with software testing, after that co-simulation and finally hardware testing on the target system. As the source of truth a MySQL build called Percona Server 8.0 was used [21], to run all queries against. Since this SQL server does not support for example half-precision floats and Arrow does support this, they are mapped to single precision floats and the code allows from some deviation and clamps any values outside the range of the 16-bit floats. Timestamps are represented as 64-bit `BIGINT UNSIGNED`, since the native `TIMESTAMP` column is 32-bit in MySQL. The full type mappings for both MySQL and software testing are in Appendix A. In figure 4.2 these two tests represent the left 2 columns of the flow chart in the middle, these are further described in section 4.5.1.

The third column from the left, the co-simulation flow, is described in section 4.5.2. The right most column represents the final co-simulation and hardware testing, this is described in section 4.5.3. And finally all the outputs are compared to the source of truth (the left column, MySQL) to see if the process works correctly.

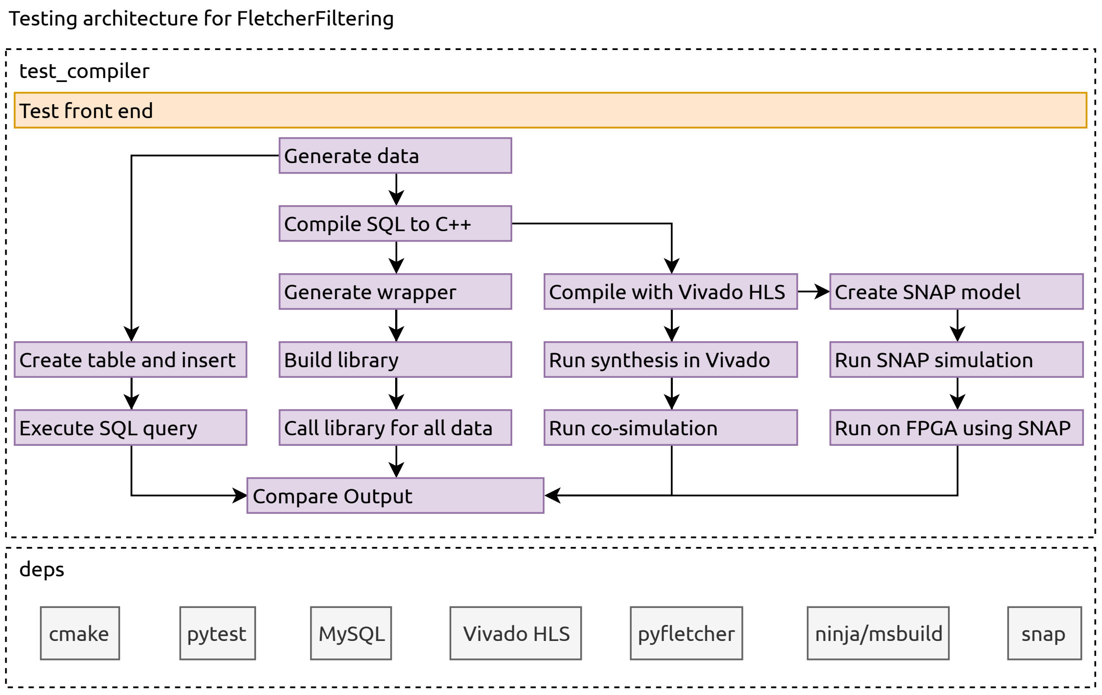


Figure 4.2: Testing Architecture

4.5.1 Software

To test the functionality of the generated C++, initially the code was compiled using regular host compilers for use with the Python package `ctypes`, this allows Python code to call dynamic libraries that use the C calling convention directly. This can speed up development a lot on platforms where either Vivado HLS is not supported or on hosts where it is not installed. Running the C++ code directly can quickly verify that there are no logic and memory access errors in the generated code. The next step is running the C Simulation in Vivado HLS, this loads a lot of extra libraries to emulate the behavior of the code on the final hardware better. This is generally gives the same output as the direct C++ runs. Note that this does not actually synthesize any RTL.

4.5.2 Co-simulation

Co-simulation is a highly useful feature of some HLS packages where the test bench can be written in C++ too, while still running the actual kernel in an actual HDL simulator. This process takes a lot longer and was added later in the project life cycle to keep development speed up, this verified that the data acquisition from the input streams and output streams is done correctly. Since this actually simulates the generated RTL representation of the kernel, this reflects reality much closer and the given result for kernel latency in cycles is fairly accurate.

4.5.3 Hardware

The next step is to run the kernel with the actual generated Fletcher hardware wrapper. Currently Fletcher does not support generating the final connection and control logic for the kernels as mentioned in section 3.4. For this testing the simulation environment of the OpenPOWER SNAP framework was used [18]. This framework handles the building of the model and generation of the FPGA image; it also provides some extra communication facilities Fletcher uses internally to pass data between de FPGA and the host CPU.

For hardware testing, a cloud-based POWER8 host was used with a Nallatech 250S FPGA card, a card with a Kintex UltraScale XCKU060 speedgrade −2 device from Xilinx. Better would have been POWER9 testing, but the support software for the available hardware was not ready in time. The SNAP and Fletcher frameworks do a great job of abstracting away the hardware and thus the host-side code can be exactly the same for both the simulation and the real hardware runs. So for the kernels to work on the actual hardware, there is theoretically very little effort required from the developer. Mostly just building the actual FPGA image and spinning up the POWER host and programming the FPGA board. Of course the nature of the POWER host will require building some of the support libraries and host-side software for the platform, but this is fairly easy to script, or the base image for the POWER machine could be updated, to include most of the required software and tooling; for example `cmake`, `libarrow` and `libfletcher` in this case to save time with each deployment.

5 | Testing

5.1 Queries

Representative tests for the compiler are very important for the measurement results to be useful. To this end, a set of queries were chosen to test all currently supported functions (as described in section 3.3). These queries also need to reflect semi-realistic scenarios. Note that in each query the `FROM` clause is effectively ignored. The given Arrow schemas are equivalent to an SQL table definition. The 'Notes' column will specify the primary key for the resulting SQL table and whether values are allowed to be `NULL`.

5.1.1 Simple

This query will test the maximum throughput per bit on the incoming and outgoing data buses. The resulting code will only be reading a value into a register and passing it along to the output. This is obviously not a truly useful application, but useful nonetheless, as it measures the maximum throughput for its given data width.

The query text is shown in listing 5.1 and the Arrow schemas are shown in table 5.1.

Listing 5.1: The most basic query, this just selects the primary key column.

```sql
SELECT
    `pkid`
FROM
    `in_ff`;
```

Table 5.1: Arrow schemas for the Simple query.

Name	Type	Notes
pkid	int32	primary

(a) Input schema

Name	Type	Notes
pkid	int32	primary

(b) Output schema

5.1.2 Wildcard

This query has some string columns, so this query will test the current implementation for reading strings. This should be much slower than the Simple query, since strings can only be read one byte per cycle, making latency and execution time dependent on string length.

The query text is shown in listing 5.2 and the Arrow schemas are shown in table 5.2.

Listing 5.2: This query tests the wildcard transform and string passing.

```sql
SELECT
    *
FROM
    `in_ff`;
```

Table 5.2: Arrow schemas for the Wildcard query.

Name	Type	Notes
pkid	int32	primary
string1	string	
string1	string	

(a) Input schema

Name	Type	Notes
pkid	int32	primary
string1	string	
string2	string	

(b) Output schema

5.1.3 Float

In this test, floating point math for different float sizes are tested. This will give results based on the floating point hardware available in the FPGA of choice. Floating point operations are expected to add a number of cycles to the kernel latency: for most high performance devices this should be around 2–8 cycles.

The query text is shown in listing 5.3 and the Arrow schemas are shown in table 5.3.

Listing 5.3: This query tests the floating point math in all supported sizes.

```sql
SELECT
    *,
    `half1` * 2 AS `half1x2`,
    `float1` * 2 AS `float1x2`,
    `double1` * 2 AS `double1x2`
FROM
    `in_ff`;
```

Table 5.3: Arrow schemas for the Float query.

Name	Type	Notes
pkid	int32	primary
half1	float16	
float1	float32	
double1	float64	

(a) Input schema

Name	Type	Notes
pkid	int32	primary
half1	float16	
float1	float32	
double1	float64	
half1x2	float16	
float1x2	float32	
double1x2	float64	

(b) Output schema

5.1.4 Concat

Here, string concatenation will be tested. Concatenation implies for the current implementation that the string needs to be composed in a block RAM (BRAM) before it can be written to the output, which should have a rather large impact on performance. The BRAMs are required over normal flip flops, because of the size of the strings, BRAMs allow for much more data to be stored.

The query text is shown in listing 5.4 and the Arrow schemas are shown in table 5.4.

Listing 5.4: This tests the concatenation of two incoming strings.

```sql
SELECT
    `pkid`,
    CONCAT(`string1`, `string2`) AS `concat`
FROM
    `in_ff`;
```

Table 5.4: Arrow schemas for the Concat query.

Name	Type	Notes
pkid	int32	primary
string1	string	
string1	string	

(a) Input schema

Name	Type	Notes
pkid	int32	primary
concat	string	

(b) Output schema

5.1.5 Combination1

This is the first query that tests the filtering function with a WHERE clause. Filtering implies performance will mainly be limited by input stream performance. The limiting factor will be the string column, and the concatenations. Filtering happens on two 32-bit uniformly distributed random integer columns (as described in section 5.2.1), in roughly the middle (4 and 18 respectively), so the expected pass rate of records is about one fourth. This means the output bandwidth will be greatly reduced.

The query text is shown in listing 5.5 and the Arrow schemas are shown in table 5.5.

Listing 5.5: This combines many of the previous tests and also tests concatenation with constants and has a filter as the WHERE clause.

```sql
SELECT
    `int1` + `int2` AS `int1`,
    CONCAT(`string1`, 1 << 4, 'NULL') AS `concat`,
    CONCAT('123456', `string1`, TRUE, FALSE) AS `concat2`,
    `timestamp1`,
    `timestamp2`,
    `timestamp3`,
    `timestamp4`
FROM
    `in_ff`
WHERE
    `int1` > 4 AND `int2` < 18;
```

Table 5.5: Arrow schemas for the Combination1 query.

Name	Type	Notes
pkid	int32	primary
int1	int32	
int2	int32	
string1	string	
timestamp1	timestamp[s]	
timestamp2	timestamp[us]	
timestamp3	timestamp[ms]	
timestamp4	timestamp[ns]	

(a) Input schema

Name	Type	Notes
pkid	int32	primary
concat	string	
concat2	string	
timestamp1	timestamp[s]	
timestamp2	timestamp[us]	
timestamp3	timestamp[ms]	
timestamp4	timestamp[ns]	

(b) Output schema

5.1.6 Combination2

This query expands on the previous one, with some in-compiler constant to string generation for floating point numbers, as well as some of the integer operators. Again, since this is a filter, this will be mainly be limited by input stream performance, with the limiting factors being the string column and concatenations. For this filter the expected pass rate of records is about one eight, because three uniformly distributed columns (see section 5.2.1) are filtered in the middle of their range.

The query text is shown in listing 5.6 and the Arrow schemas are shown in table 5.6.

Listing 5.6: This tests the comparison operators and constant generation for floating point numbers.

```sql
SELECT
    CONCAT(0.5, `string1`, 0.3) AS `concat`,
    CONCAT(3.453345, `string1`, 3.12E4) AS `concat2`
FROM
    `in_ff`
WHERE
    `half1` > 0 AND `float1` > 0
        AND `double1` > 0
```

Table 5.6: Arrow schemas for the Combination2 query.

Name	Type	Notes
pkid	int32	primary
string1	string	
half1	float16	
float1	float32	
double1	float64	

(a) Input schema

Name	Type	Notes
concat	string	
concat2	string	

(b) Output schema

5.1.7 Nullable

This is the first query that tests nullable columns and the extra control logic required for it. Some of the basic operators and functions are also tested for proper `NULL` propagation. For example, most operators will result in a `NULL` value when any of the inputs are `NULL`.

The query text is shown in listing 5.7 and the Arrow schemas are shown in table 5.7.

Listing 5.7: Query that tests the functionality around nullable columns.

```sql
SELECT
    `pkid`,
    `nullint`,
    `string1`,
    `pkid` + `pkid` AS `pkid2`,
    `nullint` * 2 AS `nullint2`,
    CONCAT(435, `string1`, 123) AS `concat`
FROM
    `in_ff`;
```

Table 5.7: Arrow schemas for the Nullable query.

Name	Type	Notes
pkid	int32	primary
nullint	int32	nullable
string1	string	nullable

(a) Input schema

Name	Type	Notes
pkid	int32	primary
nullint	int32	nullable
string1	string	nullable
pkid2	int32	
nullint2	int32	nullable
concat	string	nullable

(b) Output schema

5.2 Data

To run the queries specified in section 5.1, input data is required. To make sure every aspect is tested, this data should be sufficiently random. The queries are essentially fuzzed with random data. When done with a sufficiently large dataset, this should expose any problems with specific values.

5.2.1 Numerical

To generate the numerical values, a uniform distribution is sampled along the numerical types' full range. The only exceptions to this are the 64-bit unsigned integers and 16-bit floats. Unsigned 64-bit integers for timestamp columns are limited to 63 bits, because of an issue in the Python Arrow bindings for generating output files: these will not take unsigned 64-bit integers for timestamps. As 16-bit floating point numbers are not natively supported by Python, they need to be manually truncated to get the correct precision. Otherwise, they will have the correct range but their precision will still be 32-bit.

5.2.2 Textual

The length of the used strings is at most half of the maximum string length, so that their concatenation result will still fit in the buffers. For the actual text data, the well known lorem ipsum sample text based generator `lipsum` is used to generate random sentences. This text is commonly used in publishing and graphic design for place-holder text. The original text is based on a lightly scrambled piece of text out of *De Finibus - Cicero* and looks like mock Latin and the random generated sentences fairly accurately represent a natural language. It also completely fits into the lower ASCII range.

While further testing with full Unicode test data would be preferential, for now that is outside the scope of this thesis. Due to the nature of the test implementation, which relies on the `strlen` C function, most of it should just work, since it handles all strings as simple sequences of bytes and so does the hardware. The problem comes from the implementation of any other string processing functions, these require Unicode awareness to function properly.

6 | Results

6.1 Simulation

For the main simulation results the designs were run using Vivado HLS co-simulation. This gives latency estimations in number of cycles, which will allow us to estimate the maximum performance of the kernels. All of the numbers are the result of runs with the default clock period target of 10 ns in Vivado HLS version 2019.1. Tighter timings are therefore possible, at the cost of extra registers, though the number of cycles might become higher if there is some hardware bottleneck, such as a floating point unit.

In table 6.1, the estimated amount of clock periods and kernel latency are displayed. As expected, string processing in the current implementation has a large negative impact on the overall latency and interval. In the current implementation, strings are processed in serial fashion, so when the strings get longer, the row latency goes up. Also due to some HLS limitations for the current implementation, there is no pipelining in between records, causing latency and interval to be mostly the same. This is a mayor cause for the lower throughput numbers for some of the kernels, as shown later in this chapter. Thankfully, as shown in table 6.2 all of these kernels are rather small, so that will allow many instances on the FPGA.

Table 6.1: Estimated clock periods, average latency and interval for all kernels. Maximum input string length was 127 characters.

Kernel	Period		Latency		Interval	
Simple	4.530	ns	0	cycles	1	cycles
Wildcard	4.036	ns	887	cycles	887	cycles
Float	8.419	ns	6	cycles	6	cycles
Concat	5.172	ns	1333	cycles	1333	cycles
Combination1	4.337	ns	1124	cycles	1124	cycles
Combination2	4.337	ns	1290	cycles	1290	cycles
Nullable	5.172	ns	869	cycles	869	cycles

Table 6.2: Estimated area usage for all kernels. Maximum string buffer length was 255 characters, for an input length of 127 characters. The maximum number of kernel instances is without any place for the interconnects, but it should paint a picture of the size of these kernels. The target device's totals are listed in table 6.3.

Kernel	BRAM_18K	DSP48E	FF	LUT	Max Inst.
Simple	0	0	33	59	≈ 5.6k
Wildcard	2	0	433	990	≈ 330
Float	0	16	912	555	≈ 120
Concat	3	0	511	1511	≈ 210
Combination1	5	0	1209	2627	≈ 120
Combination2	5	0	1275	2776	≈ 110
Nullable	3	0	715	1761	≈ 180

To measure the impact of the maximum string length, the queries that have string columns were tested again with a maximum string length of 32 rather than 127. The results in tables 6.4 and 6.5 show that when the string buffers are small enough and there are no constants to store in BRAMs, the BRAMs are replaced by registers, hence the gain in used flip flops and lookup tables for most queries. This also shows that the limiting factor in the current implementation for much larger strings is the number of available block RAMs on the used device.

Table 6.3: Target device Kintex UltraScale XCKU060 total resources

Resource	Available
BRAM_18E	2160
DSP48E	2760
FF	663 360
LUT	331 680

Table 6.4: Estimated clock periods, average latency and interval for all kernels with string columns. Maximum input string length was 16 characters, with the effects indicated.

Kernel	Period			Latency			Interval		
Wildcard	2.963	ns	▼	118	cycles	▼	118	cycles	▼
Concat	5.194	ns	≈	182	cycles	▼	182	cycles	▼
Combination1	4.359	ns	≈	175	cycles	▼	175	cycles	▼
Combination2	4.359	ns	≈	217	cycles	▼	217	cycles	▼
Nullable	5.194	ns	≈	130	cycles	▼	130	cycles	▼

Table 6.5: Estimated area usage for all kernels with string columns. Maximum string buffer length was 32 characters, for an input length of 16 characters. With the effects indicated. The maximum number of kernel instances is without any place for the interconnects, but it should paint a picture of the size of these kernels. The target device's totals are listed in table 6.3.

Kernel	BRAM_18K		DSP48E		FF		LUT		Max Inst.	
Wildcard	0	▼	0	=	459	▲	986	▼	≈ 330	=
Concat	0	▼	0	=	550	▲	1517	▲	≈ 210	=
Combination1	2	▼	0	=	1248	▲	2644	▲	≈ 120	=
Combination2	2	▼	0	=	1314	▲	2806	▲	≈ 110	=
Nullable	1	▼	0	=	741	▲	1760	▼	≈ 180	=

6.1.1 Observations: Simple

The simple kernel gets almost completely optimized away since there is basically no real logic in the data path. Removing most of the extra scaffolding code results in essentially the code displayed in listing 6.1.

Listing 6.1: Simplified generated C++ code for the Simple query.

```cpp
// Read the value from the input stream
(ff_in.pkid >> pkid);
//Start of data processing
f_int32 pkid_o(pkid);
bool __pass_record = true; // The absence of the WHERE clause results
    in a straight 'true' value
//End of data processing
if (__pass_record) {
    // Write the value to the output stream
    (ff_out.pkid << pkid_o);
}
```

The actual data processing code here does nothing but create a copy of the input value to the output. In the resulting HDL, this copy will be optimized away to essentially a directly wired connection, as shown in listing 6.2.

Listing 6.2: Simplified generated VHDL code for the Simple query.

```vhdl
-- Connect the input signals directly to the output
ff_out_pkid_V <= ff_in_pkid_V;
```

Obviously, the control code to handle all the stream handshakes and other auxiliary signals is retained, but the data flow is very straight forward.

6.1.2 Observations: Queries with strings

To see what the full relation is between maximum string size and resource use and latency, the Wildcard query was tested with 127–1023 character buffers in addition to the previous tests. The used resources stay relatively constant, the only changing quantities are the used BRAMs: for a max length of 255 it stays at 2, probably until the 18 kbit of the block RAM is filled. The most important change is in estimated clock period and simulated latency. The clock period drops by about 1 ns when character buffers of 127 characters and under are used, because then BRAMs are no longer required, thus removing the control logic that adds the extra delay.

Figure 6.1 shows the relation between string length and latency. For any sufficiently large string size this relationship is linear. The non-linear part occurs when the number of BRAMs changes. To test this theory for more than 2 BRAMs, a test was run with 8095 characters. This resulted in Vivado HLS generating a design with 12 BRAMs, and a latency of 16 786 cycles, while the expected latency, calculated with a linear regression on the plotted data where the design uses BRAMs, would be around 18.5 kcycles.

This shows that the expected latency converges to $4 \cdot x + 480$ for larger strings. A test with 16 383 character buffers further confirms this relation.

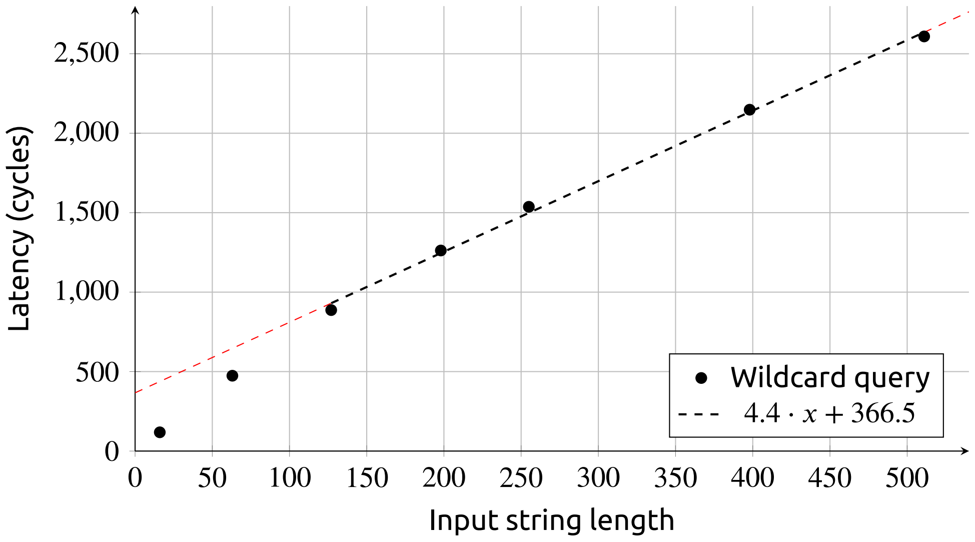


Figure 6.1: Latency and interval of the kernel with different maximum string lengths, the input string length is at most half the maximum buffer length.

6.1.3 Observations: Floating point numbers

As mentioned, for queries with floating point numbers, performance is limited by the latency of the hardware floating point units or the digital signal processors (DSPs), whichever is used, on the target device. On the used target device (Kintex UltraScale XCKU060) this results in a maximum latency of 6 cycles for a column with a double-precision multiply. On FPGA architectures with hardware floating point units, this can be reduced, or, if the DSP slices are less advanced, this might increase.

6.2 Hardware

As described in section 4.5.3, the hardware tests were run on a cloud instance. Due to hardware and software availability, only one of the queries could be tested on actual hardware. Using the knowledge gained from these measurements, the maximum performance of the other queries can be estimated.

6.2.1 Simple

The kernel was tested multiple times using record batches of 40 MB (10 Mrecords) and 1 GB (250 Mrecords). The average dataset setup and read time for the 1 GB was

134 ms, and the average clean up and output saving time 32 ms. The same values for the 40 MB record batch are 9.3 ms and 0.30 ms, respectively. With the platform setup and queueing of the record batches for both taking less than 2 ms it's basically insignificant for the larger data sets, though for very small record batches one might want to queue a larger number of them at one time.

For the smaller data sets, the standard deviation was about 100 times bigger than for the larger one, at about 46 MB/s run-to-run, so the larger data set measurements are more useful and repeatable. The measured throughput numbers for the kernel are shown in table 6.6. With the FPGA running at 250 MHz in the test system, these results indicate that the estimated interval from section 6.1 is fairly accurate for this kernel. Barring any other bottlenecks, widening the bus, like with a column of 64-bit integers, should yield a near perfect doubling of throughput. This is due to the highly parallel nature of this specific kernel and how FPGAs implement these.

Table 6.6: Throughput numbers measured on POWER8 using a Nallatech 250S FPGA card and the SNAP framework.

Kernel	**40 MB record batch**				**1 GB record batch**			
Simple	863	MB/s	216	Mrecords/s	948	MB/s	237	Mrecords/s

6.2.2 Implementation results

Due to some hardware availability issues and hardware bugs, the other kernels were not able to be run on the actual hardware. However, using the knowledge gained from the Simple test and timing reports generated by Vivado, the possible maximum throughput can estimated fairly accurately. All numbers in this section were run with a clock period target of 4 ns to match the target architecture's FPGA frequency of 250 MHz. The results are shown in tables 6.7 and 6.8.

Table 6.7: Post implementation kernel statistics, and input bandwidth potential at F_{max} and at 250 MHz, the given interval number is the reported average. The bandwidth and record size calculations for string columns use the average string length and 4 B for the string length itself.

Kernel	Period		Interval		Size		Potential		@250 MHz	
Simple	1.513	ns	1	cycles	4	B	2644	MB/s	1000	MB/s
Wildcard	2.318	ns	890	cycles	195	B	95	MB/s	55	MB/s
Float	3.425	ns	11	cycles	18	B	478	MB/s	409	MB/s
Concat	2.983	ns	1347	cycles	198	B	49	MB/s	37	MB/s
Combination1	3.209	ns	1133	cycles	139	B	38	MB/s	31	MB/s
Combination2	2.676	ns	1291	cycles	112	B	32	MB/s	22	MB/s
Nullable	2.399	ns	870	cycles	94	B	45	MB/s	27	MB/s

Table 6.8: Post implementation kernel resource usage, all of these numbers are less than 0.45 % of the total on the target device, the totals for each resource are shown in table 6.3. The maximum number of kernel instances is without any place for the interconnects, note how the number of duplicated kernels is much higher than the ones based on the estimated resource usage, because the implemented resources are much lower.

Kernel	BRAM_18K	DSP48E	FF	LUT	Max Inst.	Max Total Bw	
Simple	0	0	0	1	N/A	N/A	
Wildcard	2	0	391	420	≈ 780	43	GB/s
Float	0	13	730	202	≈ 210	86	GB/s
Concat	3	0	514	623	≈ 530	20	GB/s
Combination1	5	0	1101	1117	≈ 290	9	GB/s
Combination2	5	0	1001	933	≈ 350	8	GB/s
Nullable	3	0	651	571	≈ 580	16	GB/s

The period for the Simple kernel is the smallest possible on the −2 speed grade of the target device (specified as 661 MHz [24, page 8]).

The 250 MHz calculated bandwidth of 1000 MB/s also confirms the measured bandwidth from section 6.2.1.

For the Float kernel, the output bandwidth will be higher than the input bandwidth, due to the increased number of output columns, at 849 MB/s and 727 MB/s for the potential bandwidth and the bandwidth at 250 MHz, respectively.

The output bandwidth will be more than double the input bandwidth for the Nullable kernel, because it duplicates the columns.

In the Concat kernel, the bandwidth is mostly limited by the output bandwidth, since there is a string concatenation, implying strings are longer than the input.

Finally, the Combination kernels have a lower output bandwidth, since they drop a couple of columns and have fairly aggressive filters. They respectively only let through one quarter and one eighth of the input data, assuming the numerical values are uniformly distributed as described in section 5.2.1.

The differences in potential output bandwidth are shown in figure 6.2. These values are per instance. The maximum number of instances is much larger than the ones calculated from the estimated resource usage, because the actual synthesis optimizes the design a lot and this results in much lower area utilisation. The "Max Total Bw." column shows the potential full system bandwidth if the FPGA was filled with instances without any interconnect overhead, and the FPGA runs at 250 MHz, so some of these can go a lot higher if the frequency can go up. In the real world, most likely the achievable total number of instances is about half.

With CAPI 1.0 reaching a usable bandwidth peak limit of about 8 GB/s, every kernel can potentially saturate system bandwidth. And in the real world, the ones running on primitives will reach system bandwidth even for CAPI 2.0 which doubles that bandwidth to 16 GB/s, while the kernels with string columns will lag behind. The same holds for OpenCAPI, the next generation version of this interface should provide 25 GB/s. In the real world, the achievable total number of instances is most likely about half.

These numbers show that the currently fairly naive string implementation clearly needs some work. For floating point and string operations, it might be valuable to do all filtering on other columns first and do the operations that take a long time in a secondary kernel after the first one. This way, the higher latency has a much smaller impact on overall throughput, especially if the filter function discards many records.

To increase throughput, one can use multiple kernels next to each other. To do this, effectively one needs to process multiple record batches at the same time. Especially when the Cerata library adds kernel parallelization, the FPGA can accommodate many kernels, and since Fletcher is designed to saturate the memory bandwidth of devices, the total design should be able to hit that limit too. But first the string implementation should take advantage of the multiple elements per cycle feature of the Fletcher framework. Because the relation between latency and string length is linear, a doubling from 1 to 2 elements per cycle might already almost halve the slope of the graph as shown in figure 6.1. And since data widths of 64-bits are not a problem for the framework, 8 elements per cycle should be possible, at the cost of increased resource

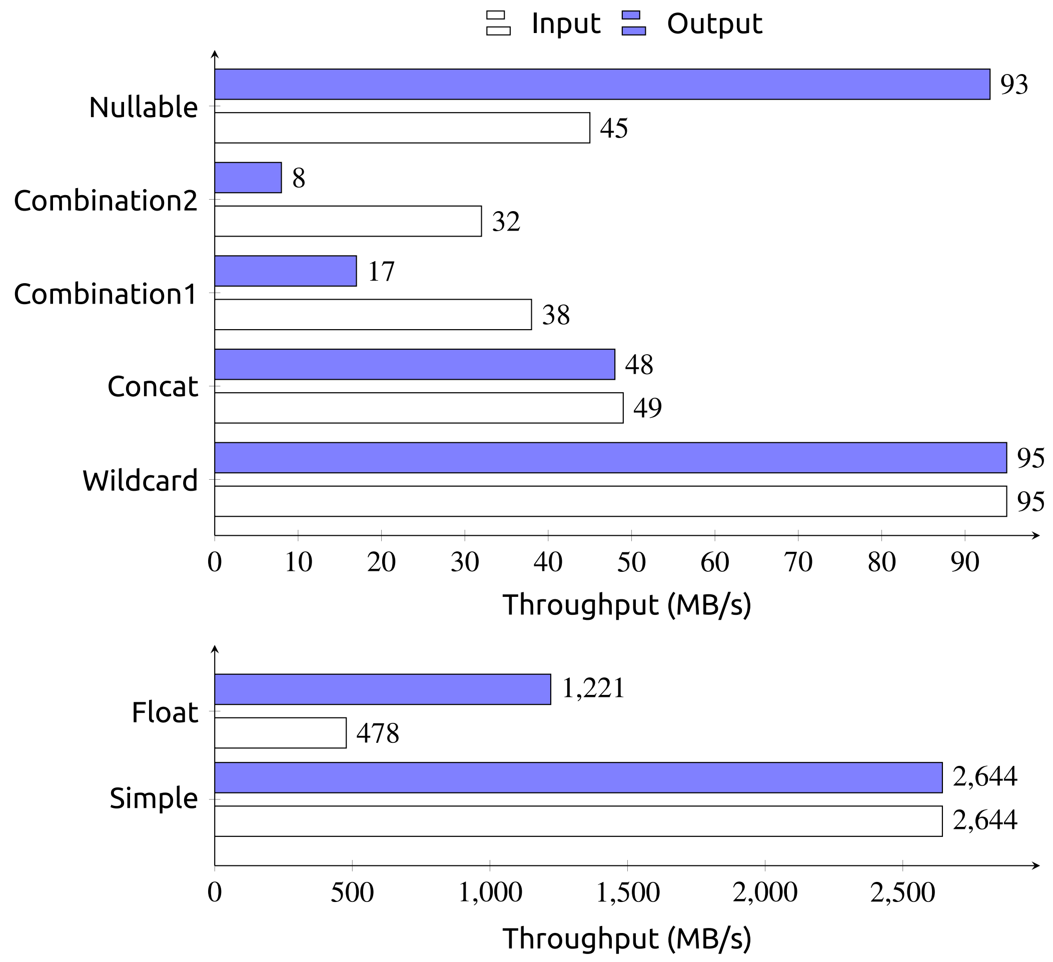


Figure 6.2: The input and output throughput of the kernels, split into two graphs. These numbers are per instance.

usage. This will however be more efficient than running multiple kernels at once area-wise. The Fletcher framework has an example called `stringwrite` which uses 64 elements per cycle and it reaches 10 GB/s for one instance [20]. With additional optimizations in the string handling, like not serially reading then writing, this means that the string column containing kernels can most likely also be made to reach the target system bandwidth.

Comparing the throughputs in table 6.7 to some of the earlier work, for example Glacier [15], does 1.6 GB/s on 3 32-bit integers and one fixed size 4 character string with a simple integer operator as a filter. This projects automatically generated kernels can reach similar speeds (2 GB/s on a similar bus width) when using fixed length strings, with something like a binary type of 32-bits in Arrow. For Centaur [19], the authors never directly mention and data throughput. In one of their tests they mention a query (`MulAdd-Perc`, a query with a multiply-add operation and a percentage calculation) that works on a single numerical column, that the hardware processes 641 queries per second on a table with 1.25M values. So an estimate for their attained bandwidth is 3.2 GB/s for this query, and that while using about 2200 LUTs for their compute operators, plus about 4 kLUTs for their scheduling, these are included in our kernels by the HLS tool. Assuming that the numbers are floating point and using the numbers from the Float kernel, to match this performance 7 instances would be required. For integers about 3 instances, both would have a smaller number of LUTs used.

7 | Conclusions & Recommendations

7.1 Conclusions

In this thesis the merits of using SQL and HLS for automatically generating filter and simple transformation kernels are discussed. The results are very promising for using the generated kernels as glue logic between larger kernels, since especially simpler filtering and transformations do not have a very large latency impact. These are therefore not affected by the number of columns, so it scales very well to wider queries. HLS tooling still has quite some way to go, but can be a great tool in the toolbox of any hardware designer, and should be considered as a tool worth looking at. The results in chapter 6 clearly show that the input C++ code for the HLS tool is still very important, even if it might perform very well as software on a CPU, writing the same code for HLS requires extra care by the developer. The results also show that, this project can match or exceed the throughput of earlier work.

In the following we will reflect on the research questions and requirements discussed in chapters 1 and 3, and provide answers to them based on the findings in this thesis.

How much of a performance bottleneck would HLS tools introduce into the generated hardware?
Depending on the situation, HLS tools generate as good a piece of hardware as the input is suitable for the tool. This means that HLS tools, with some tweaking to pragmas and other tool specific constructs, can be used to create high performance designs. In this project's instance, anything to do with longer strings makes the resulting hardware nigh unusable, though this is fixable by spending some time to change the string implementation in the code generation. By taking advantage of the multiple elements per cycle capabilities of the Fletcher infrastructure, string processing can reach the memory bandwidth of the target device, like the included example with the Fletcher framework [7]. This reduces the latency of the kernels to such an extent that they should also saturate the system bandwidth, meaning the requirements in this thesis are fulfilled.

Can HLS C++ be used as an orchestration language for hardware?
With the new developments on HLS tools and the addition of blackbox C functions to, for example, the latest version of Vivado HLS, C++ really can become the Python of the hardware world. One would only write HDL for high-performance compute kernels and let HLS deal with all the top-level logic and data movement. Potentially, Fletcher could be made into a bunch of C function prototypes that can be used in HLS to use the full power and performance right from the HLS kernels, without the use of any external tools or the need to write any HDL.

Can one automate most of this HLS code writing for filtering and simple transformations?
The current project automates most of the generation of the required code. It generates the HLS kernel, most of the other project and helper files. It can also generate a SNAP project including a kernel wrapper.

Does this automation result in saved developer hours?
Yes it does, though it is important to note that the time saved by using HLS might be offset by the added complexity of yet another tool in the hardware development stack. The final synthesis, and place and route operations still require the traditional development tools and all the pitfalls that come with them. Fletcher as a framework for using these transformations directly with SNAP might be a little overkill and one might be better off using a row-based memory format instead. That is, however, not the power of this application, which is that this format is understood by a lot of different development platforms and is standardized, thus allowing many kernels and tools to use the same memory space directly. This allows for a tight integration of this work and other Arrow and Fletcher based projects.

The most of the requirements have been met, the front end does parse SQL and the back end outputs C++ that can be understood by Vivado HLS. The signal interfaces for the Fletcher HLS API provide all required signals on a concatenated bus. Unfortunately with the current string implementation it will be very hard to reach the system bandwidth of even a CAPI 1.0 interface with a real world and more complex design.

The project code is released under the Apache 2.0 license on GitHub at .

7.2 Recommendations

This project shows that HLS can be truly viable to generate usable hardware designs. The current implementation needs some additional work to improve performance, especially for strings, for example by taking advantage of the multiple element per cycle functionality of the Fletcher framework. When that is done supporting Arrow lists should also be fairly straightforward. Furthermore, the pipeline and data flow functions of the HLS engine should be explored more; these can greatly improve performance for more complex kernels. Finally, since the current string implementation is not very good, there is a lot of opportunity to increase the total throughput of these kernels. The process is currently completely serial, because reading and writing have to wait for each other to finish; this while the strength of FPGAs is their massively parallel nature. Fixing this problem can easily result in a gain of x8–10, lowering the number of instances required to saturate interface bandwitdh.

Glossary

Arrow

Arrow is an Apache Foundation project to create and maintain a standard for in-memory data layouts, to help with reducing serialization overhead when sharing data between different technology stacks.

AST

abstract syntax tree

BRAM

Block random-access memory (RAM) consist blocks of memory on the FPGA chip itself, these are very high performance, but are often very limited in quantity and size.

C++

C++ is a high-performance, general-purpose programming language, often used for high performance applications. It was created as an extension to the C language.

CAPI

CAPI is a high speed processor expansion bus standard, it runs on top of PCI Express. In our case it allows the FPGA to have a high speed connection to the host CPU and memory.

Cerata

Cerata is a library for high-level structural hardware design. It's used internally by Fletcher to describe the hardware structures and their connections.

DSL

domain-specific language

DSP

DSPs are the main provider of math functionality in most FPGAs, they are the core in the implementations of most math functions, like multiplication and any floating point math.

FF

Flip flops or latches are small 1 bit memories in the FPGA fabric. These are the fastest way of saving data.

Fletcher
: Fletcher is a framework that brings Arrow to FPGAs.

FPGA
: Field-programmable gate arrays are large reconfigurable integrated circuits with large arrays of switches, memories and look-up-tables to allow for the creation of any hardware function.

HDL
: hardware description language

HLS
: High-level synthesis is the automated process of transforming a behavioral description written in a high-level language, like C++, into a digital hardware representation that implementents the same behavior.

HPC
: high-performance computing

IR
: intermediate representation

LUT
: Lookup tables are the main logic ingredient in an FPGA fabric. These can be used to implement any logic function, these together with all the interconnects are the brain of the FPGA.

maps to ($\mapsto$)
: This operator is used to signify the result of a transformation.

MySQL
: MySQL is an open-source relational database management system.

Percona Server
: Percona Server is a drop-in replacement fork of the MySQL database server. This is the database server used as the source of truth for the tests done in this thesis.

POWER
: POWER is a microarchitecture and brand name created and used by IBM for their microprocessors.

Python
: Python is an interpreted, high-level, general-purpose scripting language, used in anything from web applications, network administration, machine learning and generic scientific programming. With a focus on code readability.

RAM
: random-access memory

record batch
: A record batch is an Arrow data set with a certain schema.

RTL
: register-transfer level

schema
: A schema is an Arrow object that specifies how the data looks and what values and their types record batches with this schema will contain.

SNAP
: SNAP is a framwork maintained by the OpenPOWER foundation and IBM, created to facilitate the quick creation of FPGA based applications on POWER systems.

SQL
: SQL is a domain-specific language, designed to manage relational data. For this thesis' project it is used to describe the stream or filter operation the user want to create.

transpiler
: Transpilers, or source-to-source compilers, are programs that read source code written in one programming language, and produce the equivalent output in another.

VHDL
: VHDL is the main HDL that is used in this project and the projects it depends on, it is the language that describes all the different hardware structures.

Vivado
: Vivado the brand name used by Xilinx to market their software suite for synthesis and analysis of HDL designs. This software is used to turn the description of the hardware the designer creates into a bitstream that can be loaded on the FPGA hardware.

List of Figures

List of Tables

Bibliography

[1] *Apache Arrow*. `https://github.com/apache/arrow`. Apache Project (cited on page 5).

[2] *Apache Arrow Homepage*. Apache Project. url: `https://arrow.apache.org/` (cited on page 6).

[3] *Apache License*. `http://www.apache.org/licenses/LICENSE-2.0`. Version 2.0. Apache Software Foundation, Jan. 1, 2004 (cited on pages 9, 12).

[4] M. Bysiek. *Transpyle: HPC-oriented transpiler for C, C++, Cython, Fortran, OpenCL and Python*. `https://github.com/mbdevpl/transpyle` (cited on page 11).

[5] M. Bysiek, A. Drozd, and S. Matsuoka. "Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. Nov. 2016, pages 9–18. doi: `10.1109/PyHPC.2016.006` (cited on page 17).

[6] M. Bysiek, M. Wahib, A. Drozd, and S. Matsuoka. *Towards Portable High Performance in Python: Transpilation, High-Level IR, Code Transformations and Compiler Directives*. Technical report 38. Tokyo Institute of Technology/National Institute of Advanced Industrial Science et al., July 2018. oai: `ipsj.ixsq.nii.ac.jp:00190679` (cited on page 17).

[7] *Fletcher*. `https://github.com/abs-tudelft/fletcher`. TU Delft Accelerated Big Data Systems Group (cited on pages 5, 37).

[8] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. "Achieving High Performance with FPGA-Based Computing". In: *Computer* 40.3 (2007), pages 50–57. doi: `10.1109/MC.2007.79` (cited on page 5).

[9] E. Homsirikamol and K. Gaj. "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study". In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. 2014, pages 1–8. isbn: 2325-6532. doi: `10.1109/ReConFig.2014.7032504` (cited on page 1).

[10] D. H. Jones, A. Powell, C. Bouganis, and P. Y. K. Cheung. “GPU Versus FPGA for High Productivity Computing”. In: *2010 International Conference on Field Programmable Logic and Applications*. 2010, pages 119–124. isbn: 1946-1488. doi: `10.1109/FPL.2010.32` (cited on page 5).

[11] Mozilla Foundation. *Moz SQL Parser*. `https://github.com/mozilla/moz-sql-parser` (cited on page 10).

[12] *Mozilla Public License*. `https://mozilla.org/MPL/2.0/`. Version 2.0. Mozilla Foundation, Jan. 3, 2012 (cited on pages 9, 12).

[13] Rene Mueller, Jens Teubner, and Gustavo Alonso. “Data processing on FPGAs”. In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pages 910–921. issn: 2150-8097. doi: `10.14778/1687627.1687730` (cited on page 5).

[14] Rene Mueller, Jens Teubner, and Gustavo Alonso. “Streams on wires: a query compiler for FPGAs”. In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pages 229–240. issn: 2150-8097. doi: `10.14778/1687627.1687654` (cited on page 6).

[15] Rene Mueller, Jens Teubner, and Gustavo Alonso. “Glacier: A Query-to-Hardware Compiler”. In: *Proceedings of the 2010 international conference on Management of data - SIGMOD ’10* (2010). doi: `10.1145/1807167.1807307` (cited on pages 6, 35).

[16] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pages 1591–1604. doi: `10.1109/TCAD.2015.2513673` (cited on page 1).

[17] K. Neshatpour, M. Malik, M. A. Ghodrat, A. Sasan, and H. Homayoun. “Energy-efficient acceleration of big data analytics applications using FPGAs”. In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, pages 115–123. doi: `10.1109/BigData.2015.7363748` (cited on page 5).

[18] *OpenPOWER SNAP Framework*. `https://github.com/open-power/snap`. OpenPOWER (cited on page 19).

[19] M. Owaida, D. Sidler, K. Kara, and G. Alonso. “Centaur: A Framework for Hybrid CPU-FPGA Databases”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pages 211–218. doi: `10.1109/FCCM.2017.37` (cited on pages 6, 35).

[20] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars Theodorus Johannes van Leeuwen, Zaid Al-Ars, and H. Peter Hofstee. "Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow (To appear)". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019 (cited on page 35).

[21] *Percona Server for MySQL ®*. `https://www.percona.com/software/mysql-database/percona-server`. Percona (cited on page 17).

[22] Tiark Rompf and Nada Amin. "Functional Pearl: a SQL to C compiler in 500 lines of code". In: *ACM SIGPLAN Notices* 50.9 (Aug. 2015), pages 2–9. issn: 0362-1340. doi: `10.1145/2858949.2784760` (cited on page 7).

[23] S. Skalicky, C. Wood, M. Łukowiak, and M. Ryan. "High level synthesis: Where are we? A case study on matrix multiplication". In: *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2013, pages 1–7. isbn: 2325-6532. doi: `10.1109/ReConFig.2013.6732298` (cited on page 1).

[24] *UltraScale FPGA Product Tables and product Selection Guide*. Xilinx Inc. 2013-2016 (cited on page 33).

A | Arrow type mapping

1.1 To MySQL

Table A.1: The Arrow to MySQL type mappings used in testing.

Arrow Type	MySQL Column Type	Notes
bool	`BOOLEAN`	
int8	`TINYINT`	
int16	`SMALLINT`	
int32	`INT`	
int64	`BIGINT`	
uint8	`TINYINT UNSIGNED`	
uint16	`SMALLINT UNSIGNED`	
uint32	`INT UNSIGNED`	
uint64	`BIGINT UNSIGNED`	
float16	`FLOAT`	MySQL does not natively support half precision, the constraints are enforced in the interface code.
float32	`FLOAT`	
float64	`DOUBLE`	
timestamp[]	`BIGINT UNSIGNED`	The `TIMESTAMP` column is 32-bit in MySQL.
string	`VARCHAR`	The length of the column is the maximum string buffer size.

1.2 To HLS C++ and Python

Table A.2: The Arrow to HLS C++ types and Python struct type mappings used in testing.

Arrow Type	HLS C++ Type	Python Struct Type	Notes
bool	`ap_int<1>`	`?`	
int8	`ap_int<8>`	`b`	
int16	`ap_int<16>`	`h`	
int32	`ap_int<32>`	`i`	
int64	`ap_int<64>`	`q`	
uint8	`ap_uint<8>`	`B`	
uint16	`ap_uint<16>`	`H`	
uint32	`ap_uint<32>`	`I`	
uint64	`ap_uint<64>`	`Q`	
float16	`half`	`e`	`half` is a Vivado HLS extension.
float32	`float`	`f`	
float64	`double`	`d`	
timestamp[]	`ap_uint<64>`	`Q`	
string	`ap_uint<8>*`	`B`	The type is per character.