Санкт-Петербургский государственный университет

ТЮРИН Алексей Валерьевич

Выпускная квалификационная работа

Экспериментальное исследование применимости дистилляции и специализированного аппаратного обеспечения для обработки разреженных данных

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2020 «Программная инженерия»*

Научный руководитель:
доцент кафедры информатики, к.ф.-м.н., С. В. Григорьев

Консультант:
доцент кафедры системного программирования, к.ф.-м.н., Д. А. Березун

Рецензент:
инженер-программист, ООО «Ланит-Терком» О. В. Медведев

Санкт-Петербург
2022

Saint Petersburg State University

Aleksei Tiurin

Master's Thesis

An empirical study of sparse data processing using distillation and specialized hardware

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2020 «Software Engineering»*

Scientific supervisor:
C.Sc., docent. S. V. Grigorev

Consultant:
C.Sc., docent. D. A. Berezun

Reviewer:
Software developer, Lanit-Tercom LLC O. V. Medvedev

Saint Petersburg
2022

Contents

Introduction **5**

Problem statement **8**

1. Background **9**

1.1. GraphBLAS 9
1.2. Fusion 10
1.3. Challenges 12
1.4. Proposed solution 14
1.5. Distillation 15
1.6. FHW 16

2. Specialized hardware generation with fusion **18**

2.1. Distiller 18
2.2. FHW 19
2.3. Data types 19
2.4. GRIN 21

3. Memory system **25**

3.1. Dataflow implementation 25
3.2. SystemVerilog implementation 26

4. Testbench **28**

4.1. SystemVerilog 28
4.2. TCL 28
4.3. Sparse linear algebra library 29

5. Evaluation **31**

5.1. Methodology 31
5.2. Experiments 33
5.3. Discussion 34

6. Related Work **38**

7. Conclusion **39**

References **40**

Introduction

Linear algebra is an excellent instrument for solving a wide variety of problems by utilizing matrices and vectors for data representation and analysis with the help of highly optimized routines. And whilst the matrices involved in a vast diversity of modern applications, e.g., recommender systems [22, 3] and graph analysis [34, 4], consist of a large number of elements, the major part of them are zeros. For example, the matrix representing YouTube's social network connectivity contains only 2.31% non-zeros [21]. Such a high sparsity incurs both computational and storage inefficiencies, requiring unnecessarily large storage occupied by zero elements and many operations on zeroes, where the result is obviously known beforehand. The traditional approach to address these inefficiencies is to compress the matrix, store only the non-zero elements, and then operate only on non-zero values. Thus, the effect of matrices tending to be sparse in many applications makes the techniques of matrix compressed representation and sparse linear algebra to be an effective way of tackling problems in areas including but not limited to graph analysis [20], computational biology [9] and machine learning [10].

GraphBLAS [14] standard defines sparse linear algebra building blocks useful to express algorithms for already mentioned areas uniformly in terms of sparse matrix and vector operations over some semiring. These include, for instance, matrix/vector multiplication, element-wise operations (e-wise for short), Kronecker product, masking, i.e., taking a subset of elements that satisfies the mask or its complement, etc.. These are sufficient to express a lot of algorithms, e.g. *PageRank*, *Breadth-First-Search*, *Sparse Deep Neural Network* [7].

However, sparse computations appear to have a low arithmetic-to-memory operations intensity, meaning that the main bottleneck of sparse algorithms is the sparse representation itself. It induces pointer-chasing and presents irregularity of memory accesses. Thus, a number of optimizations were identified [46], whose aim is to reduce the intensity of memory accesses, and the one considered in this work is *fusion.* Fusion simply stands for gluing several functions into one to remove intermediate data structures,

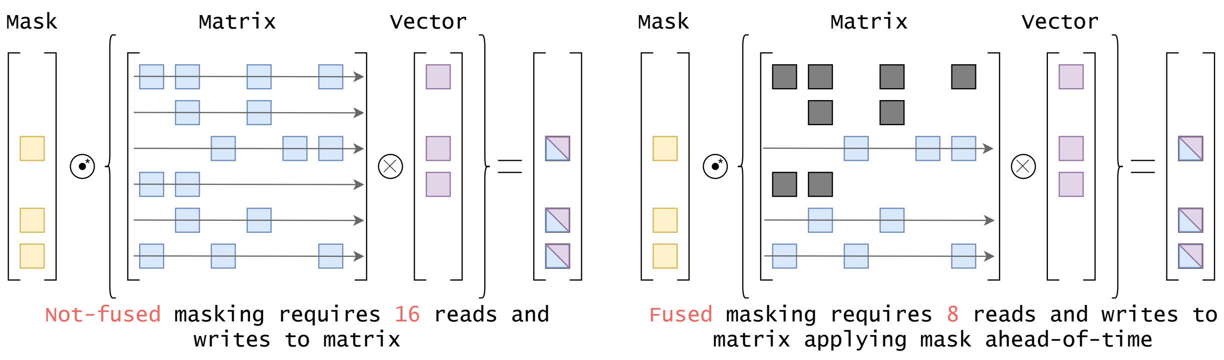
Mask
Matrix
Vector
Not-fused masking requires 16 reads and writes to matrix
Mask
Matrix
Vector
Fused masking requires 8 reads and writes to matrix applying mask ahead-of-time

Figure 1: Mask fusion

namely those that are first constructed and then deconstructed.

An example of fusion could be seen in figure 1. There a masked sparse matrix-spare vector multiplication (shown with $\otimes$) is depicted. The mask simply takes the subset of the result, but fusing the mask inside the multiplication reduces the number of memory accesses by avoiding the construction of an intermediate matrix that would store the result of multiplication.

In the case of loop-based programming, fusion simply stands for joining several loops into one to increase memory locality and reduce the number of required iterations. It is a crucial technique in dense applications and is usually implemented as a part of affine transformations and polyhedral analysis [2]. However, indexing in sparse applications is not an affine one.

Some general-purpose solutions exist that support fusion (e.g., [12]) which are based on map/reduce semantics. But in order to support sparse operations, they should be able to fuse across index arithmetic, which is not the case. Also, at the moment neither SuiteSparse [40] nor GraphBlast [46] have adopted automatic fusion in their implementations but address the issue, and these are widely used GraphBLAS implementations for CPU and GPU platforms, respectively.

Further, there are several implementations of GraphBLAS standard that show decent performance for both GPU [46] and CPU [6] backends, with one of them already speeding up graph database queries [28]. However, typical CPUs and GPUs were proven to be underutilized [8, 21, 24, 36], i.e., their computing units do not achieve peak performance, for tasks that involve sparsity, due to being too general-purpose by design and suffering from the irregularity of memory accesses incurred by sparsity.

The traditional approach for problems when existing hardware does not provide enough performance or appear to be energy-inefficient for a particular application is the design of an application-specific processor. Such an approach has found a successful application in image processing [15, 25], programmable networks [23], and machine learning [5, 45]. Thus, a possible way of addressing the underutilization issues of the current hardware running sparse algorithms is the design of an application-specific processor that could handle sparse operations more efficiently.

And indeed, there are a number of works [24, 30, 36, 37] that implement specific hardware for sparse operations. However, they generally focus on sparse matrix-matrix and matrix-vector multiplications which are not enough to express a somewhat useful subset of sparse building blocks, i.e., blocks that appear to be useful to construct, e.g., some graph algorithms like finding maximal independent subset. The low-level nature of such solutions makes it impossible to reason about them, which makes automatic fusion impossible. Finally, in such applications, the overall design energy consumption and performance are dominated by external memory accesses. Fusion reduces the number of such accesses and thus could improve the performance.

The traditional approach for developing hardware uses some low-level RTL programming, e.g., in SystemVerilog. However, high-level synthesis, a process when hardware is generated from a description given in a high-level language, makes reasoning about programs easier, opening opportunities for already mentioned fusion. Thus, the aim of this work is to compile sparse linear algebra programs into hardware with fusion optimization in mind. That will exploit the performance from the hardware side and amenability to optimizations from the software side.

Problem statement

The aim of this work is to evaluate whether it is practical and effective to utilize distillation and specialized hardware to optimize programs that contain sparse linear algebra routines. The work elaborates on distillation and high-level synthesis, outlining key challenges to obtain a better solution. In order to achieve the aim, the following objectives were set.

- Study approaches for providing fusion in different areas.
- Implement hardware generation with fusion in mind.
- Design memory interface.
- Implement the testbench and carry out the evaluation.

1 Background

The section gives the necessary insights about sparse linear algebra to more clearly show the context of the work. Further, it discusses fusion, challenges and outlines the proposed solution.

1.1 GraphBLAS

GraphBLAS is a C API specification that standardizes sparse linear algebra building blocks initially for graph computations but is nevertheless applicable in other areas. It translates mathematical specifications to API that could be efficiently implemented in hardware or software. Also, it is the only such specification and was completed by researchers from the field of high-performance graph algorithms based on sparse linear algebra. The specification further gives the means for interoperation with vertex-centric libraries, which potentially makes it a crucial component in the future ecosystem of big graphs [13]. A list of operations (not exhaustive, though) provided by the standard could be seen in table 1, where $C\langle M\rangle$ stands for masking, i.e., taking a subset of elements that satisfies the mask or its complement. These are sufficient to express a lot of algorithms, e.g., *PageRank*, *Breadth-First-Search*, *Sparse Deep Neural Network Graph Challenge* [7]. Notably, each operation is parameterizable by a semiring, which is the key to expressivity.

These operations are often chained in such a way that makes fusion possible. Consider, for example, an excerpt from Luby's maximal independent set algorithm [41] in listing 1 depicted in pseudo-Haskell with mutable variables. It shows a sequence of element-wise additions which could be fused

Function	Description	Notation
`GrB_mxm`	matrix-matrix mult.	$C\langle M\rangle = AB$
`GrB_eWiseMult`	element-wise, set union	$C\langle M\rangle = A \otimes B$
`GrB_eWiseAdd`	element-wise, set-intersection	$C\langle M\rangle = A \oplus B$
`GrB_apply`	apply unary op.	$C\langle M\rangle = f(A)$

Table 1: Some of the GraphBLAS operations

```
-- select node if its probability is > than all its active neighbors
let new_members = prob `GrB_eWiseAdd GrB_GT_FP64_Semiring` neighbor_max
-- add new members to independent set.
    iset = iset `GrB_eWiseAdd GrB_LOR_Semiring` new_members
```

Listing 1: Excerpt from Luby's maximal independent set algorithm implementation

to eliminate the construction of a `new_members` matrix in the middle. The result of such fusion depends on the implementation of `GrB_eWiseAdd` and hence is not shown.

1.2 Fusion

The formal definition of fusion is not obvious to give, and thus some intuitive understanding is assumed. Fusion optimization often stands for the removal of something intermediate, i.e., something which is first constructed and then deconstructed. These could be arrays for imperative programs, lists for functional programs, and whole kernels in the case of a GPU accelerated linear algebra library (kernel launch overhead is mitigated by gluing kernels into one). This section will give a brief overview of what is assumed by fusion in some systems and outline their implementation details. Note that only automatic fusion will be considered since the easy solution is to provide enough already fused primitives, but the approach has limited flexibility and high development costs.

In *imperative languages*, fusion is often associated with gluing several loops into one. It may increase the locality and reduce the required memory consumption. The procedure often relies on polyhedral and affine analysis techniques and is infeasible if the loops exhibit non-affine indexing, which is just the case for sparse data structures.

In *functional languages*, we mainly deal with functions, and hence fusion in its simplest form is function composition, which is implemented as a term rewriting pass, e.g., `map f . map g == map (f . g)`. However, functions are less performant in representing data than, e.g., arrays, and some state-of-the-art solutions exist that provide fusion for high-order array operations in functional languages [12]. But they also fail to fuse across index arithmetic which arises in sparse data structures. There was an attempt to implement sparse matrix e-wise addition in functional array programming language [12], but indexing appeared to be hard to fuse[1].

Furthermore, rewrite rules are domain-specific. They are tightly coupled with, e.g., arrays or matrices where one could use distributive law to perform some kind of fusion. But, in sparse applications, it is often the case that we do not have any special properties like associativity and commutativity, e.g., in the case of context-free graph querying. However, it is worth noticing that such rewrite rules are very powerful when combined with staged compilation [32]. Such rules also form the basis for a widely addressed area of *stream fusion* [38]; however sparse matrices impose a particular structure and thus are not stream-like, and rules require to express programs with certain combinators.

In Tensorflow and its XLA compiler [45] a machine learning model is represented with a dataflow graph, in which tensors flow through the edges and nodes represent some operations. The goal of the compiler is to identify and offload some parts of such a graph to, e.g., GPU, and perform corresponding code generation. Fusion is performed on that graph level, either by duplicating producers for each consumer or fusing outputs of several operations. The approach works under the restrictions that the kernels comply with similar memory access and iteration patterns, which is hard to guarantee in sparse applications.

Another approach to removing intermediate data structures in functional programs is *distillation* [16]. It is a generalization of supercompilation [39], and briefly, its goal is to represent all possible execution paths of a program with a finite process graph. Such a process graph is built by performing normal order reduction on terms, and the fusion effect is achieved by performing a transitive closure of such a graph. On top of that, distillation is able to provide superlinear programs speedup, perform specialization, and make other optimizations available by propagating positive information.

[1]Implementation is available here: `https://github.com/Tiltedprogrammer/impala-sparse/blob/master/futhark/sparse.fut` (online; accessed: 2022-06-07)

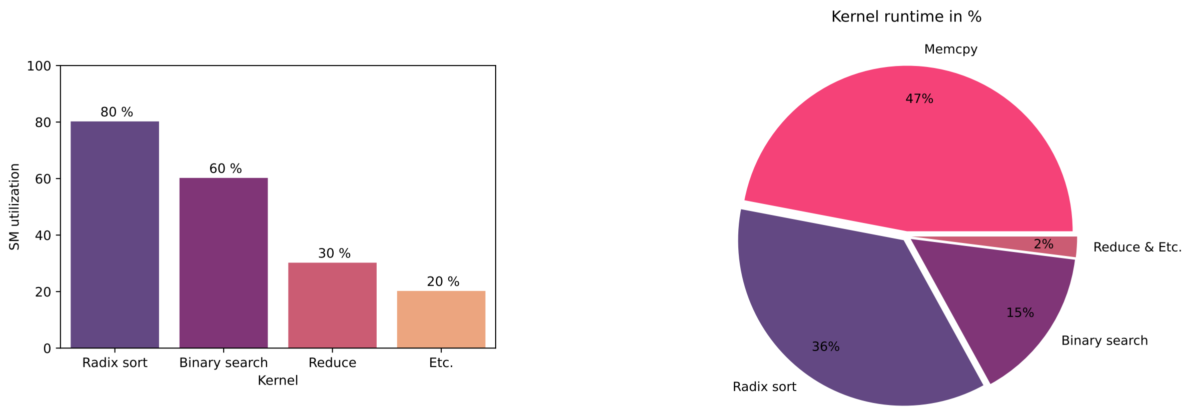

Figure 2: Resources utilization and runtime

1.3 Challenges

Despite the overall achieved performance of GraphBLAS implementations, there are inevitable challenges both for the hardware running sparse algorithms and for the software performing optimizations.

For the former, both CPUs and GPUs remain underutilized when executing sparse operations due to high cache-miss rates and limited communication between processors [24]. Sparse algorithms are inherently memory-bound, resulting in poor GFLOPs number compared to peak GFLOPs theoretically available on modern devices, namely less than 0.2% of theoretical performance is achieved as reported in [21, 8]. For example, the pipeline of sparse matrix-matrix multiplication consists of several kernels that greatly vary in performance and resource utilization. For the case of a GPU implementation[2] the performance analysis is illustrated in figure 2. Each kernel utilizes SM (Streaming multiprocessor) differently, where percentage means the number of cycles the SM is not idle. It is also worth noting that the utilization of floating-point units is still about 0% for all the kernels. Also as it could be seen from the kernels' runtime, the *reduce* kernel (the one that actually performs the operations, e.g., addition and multiplication) takes about 2% of the time, while memory operations take the whole half of it. It means that full GPU power is needed only in 2% of the pipeline time.

For the latter, optimizations are hard to automate and perform in general. Kernels are compelled to satisfy certain restrictions to be fuseable:

[2]Using cuSPARSE library and roadNet-CA graph.

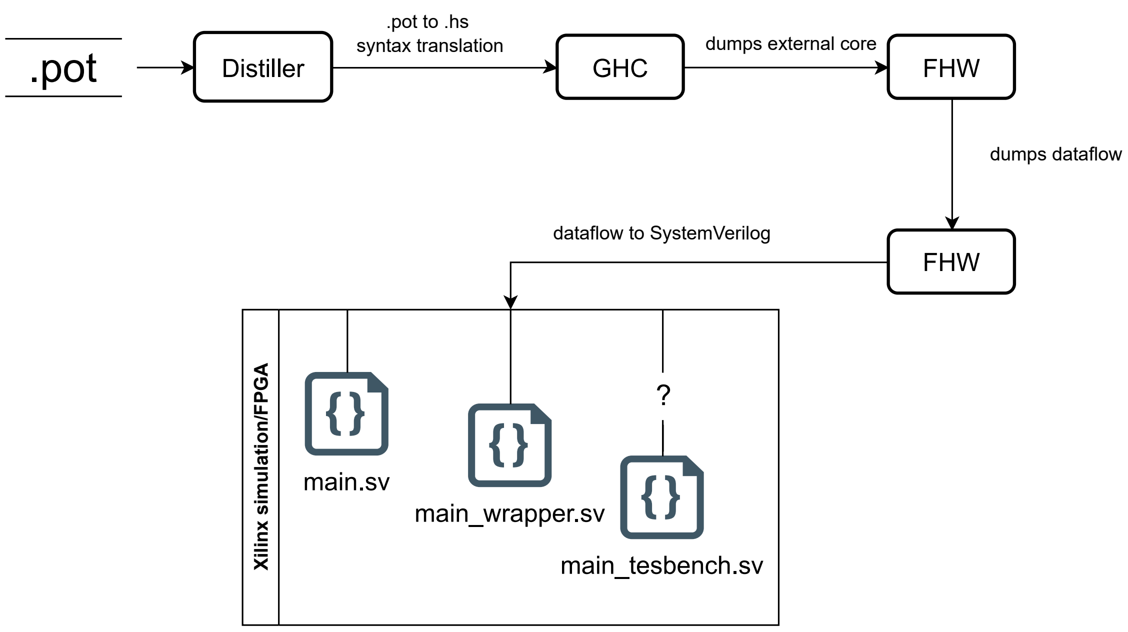

Figure 3: Solution's data flow

the absence of intermediate synchronization, same data access pattern, and enough resources on the device to execute the fused kernel. For specialization, it is better for the underlying hardware to be *MIMD* for the workers to be completely independent, however GPU's architecture is *SIMD*, which prevents successful specialization in many cases.

Eventually, some application-specific integrated circuits were designed to address the issues mentioned above, that basically provide hardware units for sparse matrix-matrix or matrix-vector multiplication [24, 35, 36, 37]. A brief overview could be found in [36]. The implementation from [36] greatly outperforms CPUs (Intel MKL[3]) and GPUs (cuSPARSE[4]) solutions in terms of speed and power consumption. Despite high performance, such solutions do not yet provide a complete implementation of GraphBLAS (or even a subset required for a concise breadth-first search) and seem too self-contained to split the operation into phases that could be optimized in the discussed sense. However, considering high performance, it is promising to combine software and hardware parts to achieve both performance and expected usability inherent to GraphBLAS implementations.

[3]Intel MKL library: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html (online; accessed: 2022-06-07)

[4]https://developer.nvidia.com/cusparse (online; accessed: 2022-06-07)

1.4 Proposed solution

To address the mentioned challenges and support automatic fusion for sparse applications, the work opts for distillation to provide fusion and high-level synthesis (HLS) to generate application-specific hardware. The first option provides low-level automatic fusion for functional programs while still making it possible to eventually use rewrite rules and staged compilation[5]. The latter one takes into account all the software optimizations made to generate application-specific hardware, and hence it is more flexible than, e.g., hardcoded solutions for matrix multiplication.

Since distillation provides fusion for functional programs, it is left to perform HLS for a functional program. Modern HLS tools are imperative and do not handle arbitrary recursion. This work uses and refines the experimental compiler *FHW* from [44]. This compiler compiles an arbitrary Haskell program into SystemVerilog utilizing parallel and pipelined dataflow intermediate representation, which allows to perform syntax-directed translation into hardware.

The flow of data in our solution could be seen in figure 3. We write a program that contains sparse linear algebra routines in a simple, functional language `.pot`, which is a source language for the distiller. After distillation, the transformed program is emitted into Haskell and GHC dumps the *external core* representation of it. This representation is input for FHW. First, a program in external core representation is transformed into dataflow intermediate representation by using auxiliary syntax transformations. Then a program in dataflow representation is fed into FHW again to finally get SystemVerilog code in `main.sv`. Some auxiliary modules are also generated: `main_wrapper.sv` is needed to pass arguments to `main.sv` and `main_testbench.sv` is simulation testbench, `main_testbench.sv` is not used for synthesis and thus is marked by `?`.

[5]It is noted to justify the design choice and is not the purpose of the work.

```
append (append xs ys) zs where

 append xs ys = case xs of
  [] -> ys
  (x:xs') -> x : (append xs' ys)
```

Listing 2: Program before transformation

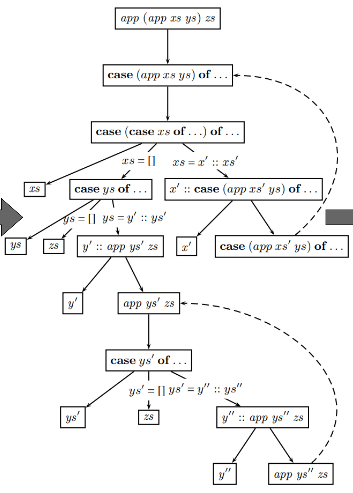

Figure 4: Process graph

```
f xs ys zs where

 f xs ys zs = case xs of
   [] -> case ys of
     [] -> zs
     (y':ys') -> y' : (g ys') where

         g ys = case ys of
           [] -> zs
           (y':ys') -> y' : (g ys')
   (x':xs') -> x' : (f xs' ys zs)
```

Listing 3: Program after transformation

1.5 Distillation

Distillation is an extension of positive supercompilation, described in [16]. It is a functional programs transformation technique that aims to reduce the number of required reductions and intermediate data structures. In doing so, it drives the program with the help of normal order reduction rules constructing a process graph. The process of driving could be potentially infinite, but it is bounded by homeomorphic embedding relation, which guarantees the termination of such driving. Such a finite process graph describes all possible execution paths of a program. Driving makes a transitive closure of such graph and propagates information, thus the number of required computations is reduced: transitive closure removes intermediate nodes, and information propagation makes extra case expressions unnecessary. An example of such transformation could be seen below, where a sequence of `append` functions is transformed to iterate over each list only once.

The process graph in the middle is obtained from driving the program to the left and the right program is residualized from the process graph. Dashed edges lead to recursive function calls, but before the call information is propagated.

Actor	Description
`primitive`	Performs built-in operations, e.g., integer addition or constructor application.
`destruct`	Disassembles a value of algebraic data type. It has an output per each data type's constructor field.
`fork`	Consumes a single input token and copies it to each of its output channels.
`demux`	Behaves like a demultiplexer, i.e. routes the value input to the channel chosen by the choice input.
`mux`	Behaves like a multiplexer, i.e. chooses the value from inputs according to the choice input and routes it to the output.
`read`	Constructs a pointer from a value.
`write`	Returns a value for a pointer.
`merge`	It chooses an available token from one of its inputs and routes it to its output.
`mergeChoice`	Same as above but also generates a token indicating which input channel provided the selected token.
`source`	Serves as an entry point to dataflow network, it is allowed to have no inputs.
`sink`	Serves as an output of dataflow network, it is allowed to have no outputs.

Table 2: FHW actors

1.6 FHW

FHW [44] is a compiler that compiles a Haskell program into SystemVerilog. Notably, the tool is purely functional and allows programs with arbitrary recursion. The compiler relies on specific syntax transformations, which make eventual translation into hardware easier. The most crucial transformations are *defunctionalization*, which encodes higher order functions, making them possible to be represented in hardware; *continuation-passing-style* (CPS) transformation, which makes control flow explicit by passing it via additional arguments to functions; and encoding recursive types with explicit pointers. CPS makes each recursive function tail-recursive, hence there is no need to maintain a stack for each function call: each group of mutually recursive functions shares the same heap, which stores their continuations. This makes it easier to return values to the caller.

The compiler relies on dataflow intermediate representation that is constructed from combinations of predefined basis actors. Then, such actors are implemented in SystemVerilog and cooperate via valid/ready protocol. Each actor is required to have at least one input and one output, thus a special `Go` token is added to trigger dataflow execution and constants creation. The available actors are summarized in table 2.

Each function and recursive data type has its own heap, which increases parallelism, but maintaining equally-sized memory spaces could be quite expensive. Memory at the moment is implemented as `bram` blocks in hardware. The memory is assumed to be garbage collected, but garbage collector is not implemented. To break cycles in valid/ready networks additional buffers are inserted.

2 Specialized hardware generation with fusion

As has already been mentioned, we utilize distillation to provide fusion for routines written in a simple functional programming language and utilize FHW compiler, which is able to translate any Haskell program into SystemVerilog and eventually into bitcode for FPGA. This section discusses how the gap between the distiller language and Haskell was closed, what issues in FHW were found and how they were solved. Also the section describes data structures being used for which fusion via distillation works well.

2.1 Distiller

The distiller from [16] operates on programs written in `.pot` language: a simple call-by-name functional language. Its syntax resembles the one of Haskell, but it encloses the arguments of a constructor in parentheses and separates them with a comma to make parsing easier. In order to emit Haskell, the pretty-printer prints such constructors correspondingly and maintains correct indentation for case alternatives, which are originally separated by `|` in `.pot`. With such modifications, it is possible to emit Haskell code from `.pot` programs and eventually pass them into FHW compiler. It is worth noting that `.pot` assumes a Hindley-Milner type system, but its implementation is untyped; thus the emitted program should be extended by hand to contain proper type definitions and annotations where needed to eliminate unnecessary typeclasses.

Another issue is that the distiller originally produced duplicated functions during residualization. Such functions increase the size of the generated hardware significantly for reasonable examples and make translation into hardware more complex. The functions are filtered to contain only non-duplicates. Since functions use De Bruijn indexes for bound variables, functions are considered to be duplicates if their bodies are equal after we replace each recursive call in one function with the name of the other function.

2.2 FHW

Unlike [11], which utilized C++ Verilator simulation, we use Xilinx Vivado to both simulate the generated SystemVerilog and to synthesize it. It allows to better estimate design and memory transferring overheads.

Initially, generated hardware did not work in Vivado simulation due to the multiple driving of `initial` blocks, which are also not synthesizable. Such blocks were replaced with a `reset` signal. The dataflow network works under valid-ready protocol, and memory output had to be invalidated to prevent the propagation of undefined signals. Such outputs are invalidated on the positive edge of the reset signal.

After that, it appeared that the compiler could not handle examples with sparse linear algebra routines due to the limited maximum number (64) of inputs in merge actors. For each merge actor in the dataflow there is a sum datatype with the number of fields equal to the number of merge inputs to be able to correctly type each merge and to identify the chosen input to pass that choice to another actor: demux, fork or mux. Such datatypes are recursively split until every datatype satisfies the restriction. The inputs of a merge are also split and a new layer of merge actors is added on top, as it could be seen in figure 5 where the limit is 2 while the original input size is 4; consuming actor is split accordingly. Similar procedure is applied for datatypes that arise during lowering passes of FHW, for example, if the defunctionalization pass introduces more than 64 constructors in the corresponding datatype. An example of such step could be seen in listing 4, if we suppose that maximum supported number of alternatives is 4.

2.3 Data types

The success of sparse linear algebra functions fusion depends on the compressed representation being used. Our choice here is quadtree representation [33]. It provides a reasonable compression rate compared to traditionally used formats, e.g., coordinate format, or compressed sparse row

```
data Cont = C_1 | C_2 | C_3 | C_4
-- transformed into
data LeftCont = C_1 | C_2
data RightCont = C_3 | C_4
data Cont = LeftCont | RightCont
```

Listing 4: Data type split step

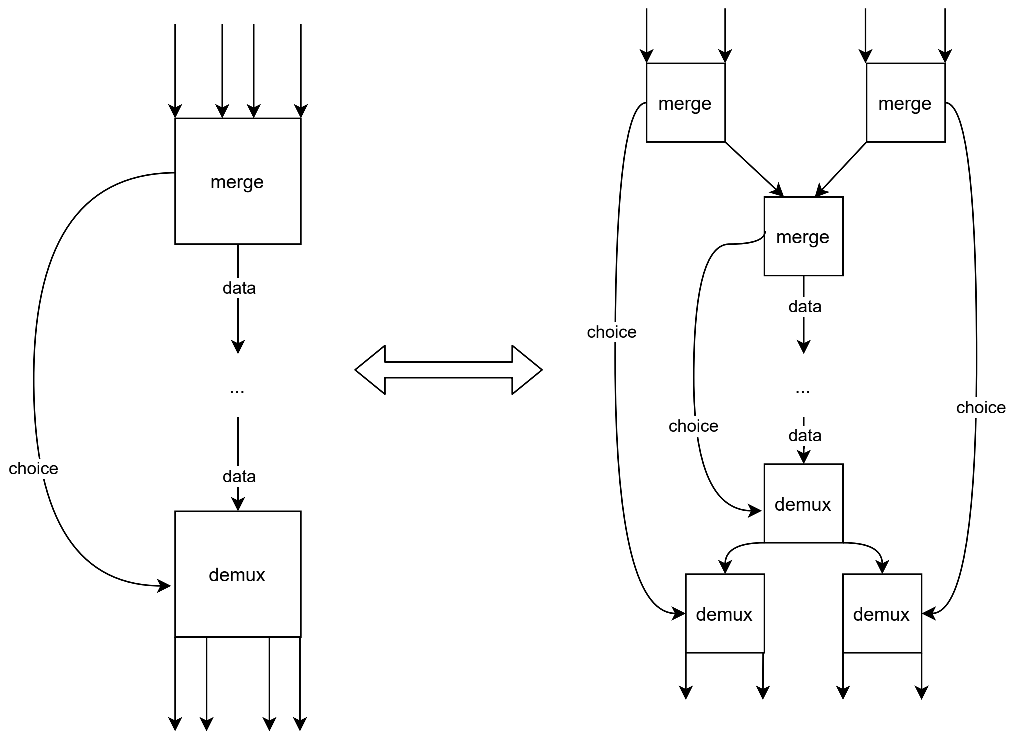

Figure 5: Merge split

representation [20], consuming 1.5x more memory on average in our experiments, although memory consumption grows logarithmically with the dimensions of a matrix. Also, it allows to express algorithms in a divide and conquer manner leaving indexing implicit, which facilitates fusion. The representation simply recursively splits the matrix into submatrices until the size of 1x1 or until the submatrix is empty. Thus it encodes only matrices with the size of power of 2, but any matrix could be extended with zeroes to fit the size requirement. An example of such representation could be seen in figure 6, where a matrix with 4 non-zero entries is depicted. Dashed squares represent either submatrix with only zero entries or zero entry. A procedure of getting a quadtree representation from coordinate format could be found in [42]. Since the representation is constructive, it is possible to utilize dependent type programming techniques to, for example, specify correct by construction routines or use rewrite rules for matrix relations to optimize routines, following the approach of [31]. Next, we use terms tree, quadtree or matrix interchangeably: they all mean a sparse matrix representation via quadtree.

2.4 GRIN

Some other issues with FHW's frontend exist. FHW uses *external core* representation of GHC, which was removed after GHC 7.6.3. This makes the compiler tightly coupled with Haskell, with the necessity to support various Haskell features, like type classes and burdens of primops. The obsolete version of GHC also makes the development process harder. FHW at the moment does not support partially applied functions in a frontend and CPS transformation is slow.

To mitigate the above issues, we opted for graph reduction intermediate notation (GRIN) [26] as a prospective frontend that would tie the frontend of the distiller and dataflow representation. It provides extra optimizations, like heap-points-to analysis, inlining, common subexpression elimination, etc., which improve the results of distillation: the distiller naively inlines some terms duplicating computations.

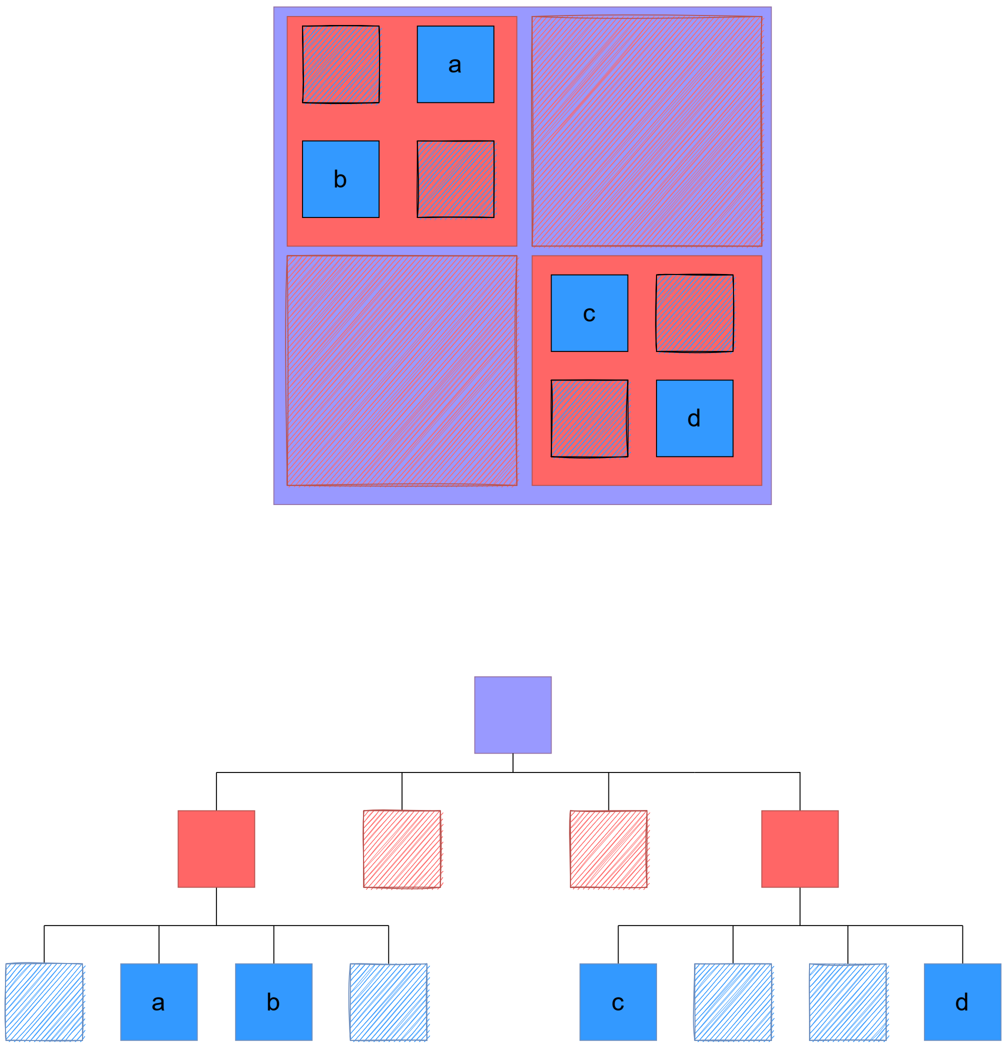

Figure 6: Quadtree representation

All the essential frontend transformations of FHW are actually parts of the translation procedure into GRIN. The procedure includes variable names unification, variable lifting, lambda lifting, and defunctionalization. These steps were implemented to translate a program from `.pot` language into `.grin`. `.pot` language does not allow recursive let bindings, which makes lambda lifting a bit easier. The algorithm could be found in [18]. Defunctionalization is a part of the GRIN itself: the representation defines `ap`, `apply`, and `eval` procedures. The first one takes a function node and an argument node: it evaluates the function node and applies it to the argument. The second one operates with partially applied functions and constructors: it either saturates a function with one more argument or makes a function call if the argument fully saturates the function. The last one simply evaluates each possible node in a program. These procedures are built during translation, when either a constructor application, or a function application (partial or saturated) are encountered. The notation is designed for lazy languages: it accumulates nodes without evaluating them if not needed, however the hardware is strict (otherwise, it would be needed to force node evaluation before transferring data from hardware to host). To make translated programs strict, calls to `eval` procedure are inserted to each function or constructor argument and functions are guaranteed to return and receive pointers to evaluated nodes. The implementation of the translator from `.pot` to `.grin` could be found at [43].

GRIN representation uses static single assignment (SSA), and whilst SSA and CPS are equivalent [19] they are different when generating hardware through dataflow. The current implementation is heavily CPS-dependent, which makes control flow in dataflow straightforward: every recursive call is a tail call. This makes parallelization harder, since otherwise, chained recursive calls could be executed in parallel instead of chaining them with continuations. Such parallelization is more explicit in SSA form. However, SSA conversion into hardware would require heavy modifications to the current dataflow scheme and it is unclear whether it will maintain the existing formalisms that dataflow fulfills. Since CPS is crucial, to express it in GRIN several auxiliary routines are added and existing ones are modified: `eval` is modified to accept a continuation; `ap` applies a function to an argument in a continuation, which is passed to function evaluation; `apCont` is a version of `ap`, which accepts a continuation; `applyC` applies a partial function or constructor to an argument with continuation; `evalC` and `evalC2` are `eval` continuations for `ap` and `apCont` respectively. A full example for CPS transformation of a program, that adds to zeroes in Peano arithmetics could be seen on GitHub[6,7].

Unfortunately, it appeared that heap-points-to analysis in the current implementation of GRIN worked only for tiny programs[8], which is not the case for our sparse linear algebra routines. Since this analysis is crucial in GRIN, the prospective frontend was left until the analysis is performant enough and the part between GRIN and dataflow representation is unimplemented at the moment.

[6]https://github.com/Tiltedprogrammer/SparseLinAlgHardware/blob/master/grin-examples/zero_plus_zero.grin (online; accessed: 2022-06-07) Original program.

[7]https://github.com/Tiltedprogrammer/SparseLinAlgHardware/blob/master/grin-examples/zero_plus_zero.grin.cps (online; accessed: 2022-06-07) CPS-transformed program.

[8]https://github.com/grin-compiler/grin/issues/128 (online; accessed: 2022-06-07) A corresponding issue related to GRIN supporting only small enough programs.

3 Memory system

Initially, FHW did not support passing arguments to the `main` function, which meant that all matrices were stored in a program's text. It increased the generated circuit size making it unsynthesizable, and did not allow to calculate all overheads, including memory transferring. This section describes a corresponding workaround, namely, it shows the transition from the listing 5 and how it was implemented at dataflow and hardware levels.

3.1 Dataflow implementation

The first step is to compile such programs with `-fno-do-eta-reduction` GHC flag to prevent GHC from doing `main = f`, since the `f` and its mutually recursive functions should be translated as a whole while `main` function is translated separately. The next step is to enumerate all the arguments to `main` in order to be able to distinguish the order of arguments of the same type when transferring data. After that, a corresponding fork node is added for each argument that "forks" each argument to the place of its usage, e.g. to the `f`'s argument.

Memories are implemented as `bram` nodes in the dataflow. All the writes/reads to the same memory are collected at a corresponding `merge` actor, which chooses the needed input, and another `merge` actor chooses whether it is write or read action. The result of read/write is sent to `demux` node, which routes the result to the caller. A new input to the merge actor

```
-- How to get from

main :: QTree Bool
main = let m_1 = ...
           m_2 = ... in
       f m_1 m_2

-- to
main :: QTree Bool -> QTree Bool -> QTree Bool
main m_1 m_2 = f m_1 m_2
```

Listing 5: Transition to `main` function with arguments

is added for each `main's` argument. Since we do not have any caller in such a case, `demux`'s output is sent to `sink` nodes. The only thing that is left is to initialize each memory's pointer. For this purpose, special dataflow `source` node inputs are added, which initialize memories for each type in `main`'s arguments, and a `sink` node is created, which shows the pointer, where the current value is stored at.

3.2 SystemVerilog implementation

Modifications from above provided us with three types of `source` nodes, one to initialize memory pointer, one to provide values to store in memory, and one to finally provide pointers for `main`'s arguments. These arguments are either scalar values (and, in that case, are not stored in any memory and just flow through dataflow edges) or quadtree node pointers (or quadtree mask pointers). Assuming memory size of 2^{16} (it models the situation when data fits into the cache, and currently, FHW supports memory spaces of sizes either 2^8, 2^{16}, or 2^{32}: 2^{16} is a tradeoff between matrix sizes and maximum available memory resources on FPGA boards) and that quadtree nodes are represented as the data type below (`QError` is needed for error checking), FHW encodes a node in the following manner: first is a valid bit, then an encoding for the node constructor (`QNone : 00`, `QVal : 01`, `QNode : 10`, `QError : 11`), and then goes either four 16 bit pointers to other nodes (in case the node is `QNode`) or the encoding for a value (32 bits for integer, for example). For example, `QVal 4` is encoded as `0{61}100100` (least significant bit is to the right).

```
data QTree a = QNone
             | QVal a
             | QNode (QTree a) (QTree a) (QTree a) (QTree a)
             | QError;
```

Listing 6: Quadtree data type

Our trees are complete trees, i.e., every node is saturated (contains four children). To serialize and then deserialize such trees, one stack is sufficient. First, we store reversed post-order tree traversal and pass tree nodes in such order to the corresponding input port of a wrapper for the dataflow module. We push a pointer for the currently stored node onto the stack (the wrapper takes it from the output port of a `sink` node added above) then if the input node is `QNode`, we pop four values from the stack and construct a new `QNode` with these four pointers in the body, pushing pointer for `QNode` onto the top. The wrapper then routes this input to the corresponding `source` node added above. The dataflow network operates under valid/ready protocol, thus we utilize AXI-stream [1] protocol to transfer trees into the circuit. The protocol makes it possible to use other predefined blocks in future FPGA design. The wrapper also makes sure that the wrapper's input is of appropriate width for the protocol, extracting the encoded node from more wide representation if needed. The end of each tree is marked with `tlast`, and the wrapper `tready` becomes invalid when the last tree arrives (that is why we enumerated the arguments). When all the arguments of all the types arrive, the circuit is finally ready.

4 Testbench

This section describes the implementation details of the testbench used to perform benchmarking. One part of the testbench is implemented in System Verilog and another part in *TCL* scripting language, which manages simulation. Also, the section describes a small sparse linear algebra library written in `.pot` language that we used for experiments.

4.1 SystemVerilog

Testbench is a generated module that interacts with the wrapper module by providing appropriate inputs to AXI-stream ports of a wrapper. It reads serialized trees from the given files and sequentially transfers nodes to the wrapper module. The file reading is performed in `initial block`, so it does not take any simulation time.

The generated dataflow circuit may optionally contain profiling registers for counting the number of memory writes or reads, namely the number of cycles `write_enable` is `1` and read's value is valid and consumer's input is ready. These registers are disabled during synthesis.

4.2 TCL

It is not possible to pass arguments to SystemVerilog module during simulation, and for this purpose, TCL language is used since a TCL script in its turn may receive any arguments. The script initializes the project and runs the simulation, providing and gathering values of interest. To modify wire values in SystemVerilog testbench `set_value` TCL command is used. It is needed to specify file paths to read the input trees from. Unfortunately, SystemVerilog does not support `set_value` for string values, thus all files are enumerated and only file number is set. To collect the information about how long data transferring takes, TCL script uses `add_condition` command, which queries the current simulation time on the event occurrence. After the simulation is ended, the script queries current simulation time once again and queries the values of profiling registers. It outputs both

Function	Description
`mAdd :: (b->Bool) -> (a->a->b) -> QTree a -> QTree a-> QTree b`	Performs an arbitrary element-wise operation (passed as the 2nd argument), between two matrices. The 1st argument defines if the resulting element is zero.
`mMask :: QTree a -> MQTree -> QTree a`	Takes a subset specified in the 2nd argument of the elements in the 1st argument. Mask is also represented as a quadtree, nodes just do not store any values.
`kron :: (b->Bool) -> (a->a->b) -> QTree a -> QTree a -> QTree b`	Performs Kronecker product of two matrices. The 1st argument defines if the resulting element is zero. The 2nd argument defines the operation applied between an element of the 3rd argument and the 4th argument.
`map :: (b->Bool) -> (a->b) -> QTree a -> QTree b`	Applies a function, specified as the 2nd argument to matrix entries from the 3rd argument. The 1st argument defines if the resulting element is zero.

Table 3: Library functions used in evaluation

time and values to the specified `.yaml` file. Thus the result of any benchmark is stored completely, which makes a basis for future benchmarks. In a similar manner other tasks could be automated using TCL scripting, e.g., synthesis and resource utilization reporting.

4.3 Sparse linear algebra library

We also developed a small library of sparse linear algebra routines in `.pot` language that we used in our experiments. The library follows the API of GraphBLAS and provides functions for matrix-matrix multiplication, element-wise operations, Kronecker product, `foldr`s and `map`s, and masking (i.e. taking a matrix's subset). At the moment, we do not focus on matrix-vector operations, hence our library does not express, e.g., finding all shortest paths. However, the functionality is enough to express other useful algorithms, e.g., triangle counting. The key to expressivity is that `.pot` is a functional language, thus all the functions are parametric with respect to matrix elements operations. `.pot` language does not have any primitives, so passing a matrix could be achieved via emitting Haskell code, or creating a module that contains a proper matrix definition of constructors and importing this module into a program. Such definitions could be obtained via utilities from [42]. Programs could be executed in several ways: by using `.pot` interpreter, using `.grin` interpreter, using Haskell emitting, or in hardware simulation.

Function signatures used in the next section are summarized in table 3. We choose these functions because we are sure the distiller yields correct and appropriate results for chains of them.

5 Evaluation

This section describes the methodology and answers the following research questions.

1. Does fusion via distillation give any benefits at the software and hardware levels?
2. What are the properties of the generated hardware?
3. Does the generated hardware outperform software implementations?

5.1 Methodology

Our focus is on creating a basis for future research and experiments, thus we make our experiments as much reproducible as possible[9]. We benchmarked the following list of chained functions. The choice is prompted by the current state of the distiller: at the moment, it does not successfully distill matrix multiplication. However, the functions are still practical enough, for example, chained addition could be seen in Luby's maximal independent set algorithm and clearly describe the applicability of the proposed approach.

- `mAdd (==False) (||) (mAdd (==False) (||) m1 m2) m3`
- `mask (mAdd (== False) (||) m2 m3) (m1)`
- `map (==Zero) (to_nat) (mAdd (==False) (||) m1 m2`
- `map (==Zero) (to_nat) (kron (==False) (&&) m1 m2`

Above, `Zero` and `to_nat` are corresponding definitions for Peano arithmetics, since the `.pot` language does not have any primitives. For the same reason, we operated with boolean matrices. Such functions could be abstracted with free variables and then instantiated in the emitted Haskell

[9]https://github.com/sedwards-lab/fhw/tree/sparse-linear-algebra-distillation/examples/QTreeBenchmarks/diploma/verilog-bool-no-nnz-inlined (online; accessed: 2022-06-07) Here one could find all the results. For each benchmark all statistics are specified: matrix names, their sizes, collected metrics for both hardware and software benchmarks.

code. However, to get maximum from distillation, we provided all the information about the functions.

For these functions, we compared the execution time of distilled and not distilled hardware generated circuits, execution time of original and distilled Haskell code and reference *Suite Sparse*[10,11] variants of these functions in C++. Note that SuiteSparse does not support recursive data types, thus only the first two function chains were implemented in SuiteSparse (since Peano number is essentially a linked list). We did not replace Peano numbers with integers, so our experiments could be interpreted easier. For hardware experiments we collected execution time and the number of memory writes and reads, to access how well fusion is performed. For software experiments we only measured the execution time. Also note that we measured only the time, required to execute the lines above, not including any IO, required to get and evaluate function arguments. But in hardware benchmarks we measured the time required to pass arguments into the circuit's memory, because such IO is inevitable. It is tricky to make such measures in Haskell due to laziness, thus the programs were compiled with `-fno-full-laziness` to turn off memoization. Also all the arguments were forced to normal form via `force` and `evaluate`. Haskell programs were compiled[12] with `-O2 -fno-full-laziness` and Suite Sparse was compiled with default flags and linked as a shared library to C++ code.

We took matrices from SuiteSparse matrix collection with sizes ranging from `64x64` to `512x512`. We limited ourselves with such sizes due to the fact that this is the maximum sizes that fit into `bram` with 2^{16} address space. Such number of `bram` blocks is available only on really expensive FPGA boards, thus in practice sizes would be smaller to achieve better utilization. Once again, it models the situation when data fits into the cache, since `bram` in our circuits will operate as a cache in real application.

[10] https://github.com/DrTimothyAldenDavis/GraphBLAS (online; accessed: 2022-06-07), Suite Sparse library sources.

[11] The library also uses different variations of coordinate formats (opaque to the user) and not a quadtree representation.

[12] GHC 8.10.4.

5.2 Experiments

Table 4 shows the results of all execution time benchmarks. To evaluate execution time for hardware simulation, implementation stage was performed to assess the maximum frequency of FPGA device used for synthesis and implementation, and the number of execution cycles was multiplied by the number of nanoseconds a clock cycle takes. The frequencies were equal within the same benchamark set, i.e., frequency was not affected by distillation. We used `xcu250figd2104-2L` device[13] for synthesis and implementation stages. It is not really a casual and affordable chip, but it contains enough `bram` for our evaluation to see scalability. In the table a median across several benchmarks is shown.

As it could be seen, distillation steadily increases performance: up to 2x speedup for hardware simulation and up to 3x for software benchmarks. The results are maintained within the borders of the corresponding confidence interval and the borders are not shown for brevity. Hardware speedup is lower due to the different execution semantics, dataflow is not reduction-based and distillation is a reduction-based transformation. Note that generated hardware appears to be less performant than both Haskell and C++, which a bit contradicts the results from [17]. For hardware benchmarks `time (IO)` shows the execution time including the time needed to transfer the data though the arguments, `time (no IO)` does not include it in its turn. It could be seen that not all the benchmarks are computationally extensive enough to cover memory transferring costs, but for more complex examples the ratio would be better. Since we basically transfer the matrices node by node from a file in the testbench, we have probably the lowest possible latency, and in practice it would be higher if reading from DDR, however the bandwidth could be increased. Noticeably, running times for `mMaskAdd` for C++ and distilled Haskell are similar, which shows that fusion really provides some extra performance: SuiteSparse at the moment does not implement any fusion.

Table 5 summarizes the ratios between distilled and not distilled hard-

[13]https://www.xilinx.com/products/boards-and-kits/alveo/u250.html (online; accessed: 2022-06-07)

ware circuits memory reads and writes. Since in general case distillation removes extra pattern matching, essentially it saves memory reads and writes. The eventual number of memory reads and writes is implementation dependent, thus the table shows what share of speedup is prompted by saving memory operations. Distillation successfully reduces the number of memory accesses, about 15% on average. `mMapKron` has a bit higher ratio due to the fact that `Nat` numbers require additional memory accesses, since the type is recursive. It could also be seen that a major part of speedups is attributed to saved memory accesses.

Finally, table 6 shows device resources utilization ratios between distilled and not distilled hardware circuits and frequencies. Columns are device primitives: registers, lookup tables, `bram` blocks or multiplexers. Utilization for both types of circuits is below 1% of available resources on the device, except for the memory. Memory blocks utilization is about 30% (since we choose larger `brams` to store larger matrices). Apart from that, distilled circuits could have both higher and lower utilization. Since the hardware generation is primarily syntax-directed it follows from the distilled program structure. For example, distillation might glue two recursive functions into one (in that case, memory utilization would be lower, because each cluster of mutually recursive functions possesses its own heap) or make more recursive functions than in the original program. The frequencies are the same, however, they possibly could be made better with more intelligent buffer allocation.

5.3 Discussion

Answering the research questions above.

1. Fusion gives significant benefits, however at the hardware level the benefits are a bit smaller since hardware semantics is not reduction based. The benefits at the hardware level are mostly determined by the reduced number of memory accesses (each access takes 2 clock cycles). Notably, distilled Haskell implementation of `mMaskAdd` has similar performance with C++.

2. Device utilization is low, but such circuits could be copied on the same device to provide better utilization and higher parallelism. Resource utilization might be both better and worse after distillation, depending on the transformed program itself since translation is syntax-directed. Frequency could be increased by more intelligent buffering strategy.

3. Although we use low-latency design with `brams` that take 2 clock cycles per request and transfer matrices from files, which does not have any latency in simulation, we get slower execution time than Haskell and C++ counterparts. It could be partly due to excessive buffering performed by FHW at the moment. Also there is no pipelining for recursive calls, i.e. only one set of function argument tokens are allowed to enter a tail-recursive function call until a result is finally generated. Further CPS transformation hinders parallelization, which could be made more explicit with SSA. Some other optimizations exist that may significantly influence the performance. Also, since device utilization is about 1%, such circuits could be copied on one device and provide more parallelism. A more detailed discussion could be found at [11].

Distillation clearly showed its applicability to optimization of sparse linear algebra routines and notably it still could be combined with other techniques, like rewrite rules to achieve better results. High-level synthesis has a room for improvements by increasing pipelining, parallelism and frequency and the generated hardware could be improved from usability perspective: a support for arbitrary sized matrices is desirable. Thus we will focus on these directions. Probably a better solution would be to embed `.pot` language into e.g. Haskell to leverage its type system (to be able to use some rewrite rules as well), and add support for primitive types and parallel primitives to be able to conduct a more scalable comparison with SuiteSparse (since SuiteSparse is multithreaded). For such embedding different execution models could be implemented, including hardware synthesis, for which SSA form of GRIN looks promising, as well as extra optimizations shipped with GRIN. For hardware synthesis, an interesting direction is achieving predictable results in hardware from certain modifications in software. This property partly holds for the current approach, since the translation is syntax- directed. More information on this could be found at [27].

mAddAdd

Matrices dimensions			Haskell	Haskell (distilled)	FHW		FHW (distilled)		C++
m1	m2	m3	time	time	time (no IO)	time (IO)	time (no IO)	time (IO)	time
64	64	64	29 us	20 us	76 us	170 us	64 us	158 us	14 us
128	128	128	94	79	146	476	134	469	30
256	256	256	123	103	202	681	168	662	44
512	512	512	219	143	474	1192	375	1093	49

mMaskAdd

Matrices dimensions			Haskell	Haskell (distilled)	FHW		FHW (distilled)		C++
m1	m2	m3	time	time	time (no IO)	time (IO)	time (no IO)	time (IO)	time
64	64	64	10 us	7 us	64 us	133 us	46 us	111 us	18 us
128	128	128	38	30	118	322	75	292	33
256	256	256	48	42	168	498	104	456	46
512	512	512	126	76	400	762	300	729	65

mMapAdd

Matrices dimensions			Haskell	Haskell (distilled)	FHW		FHW (distilled)		C++
m1	m2	m3	time	time	time (no IO)	time (IO)	time (no IO)	time (IO)	time
64	64	—	45 us	37 us	189 us	253 us	137 us	202 us	—
128	128	—	162	105	524	685	397	579	—
256	256	—	312	216	1047	1360	680	986	—
512	512	—	436	273	1346	1776	900	1330	—

mMapKron

Matrices dimensions			Haskell	Haskell (distilled)	FHW		FHW (distilled)		C++
m1	m2	m3	time	time	time (no IO)	time (IO)	time (no IO)	time (IO)	time
2	64	—	64 us	36 us	212 us	242 us	94 us	125 us	—
2	128	—	137	68	434	502	199	266	—
2	256	—	364	126	1004	1188	449	636	—
4	128	—	302	94	694	763	330	401	—

Table 4: Execution time

mAddAdd

Matrices dimensions			Ratio $(\frac{FHW}{FHW_{distilled}})$	
m1	m2	m3	writes	reads
64	64	64	1.10	1.15
128	128	128	1.02	1.05
256	256	256	1.03	1.06
512	512	512	1.10	1.16

mMaskAdd

Matrices dimensions			Ratio $(\frac{FHW}{FHW_{distilled}})$	
m1	m2	m3	writes	reads
64	64	64	1.13	1.26
128	128	128	1.06	1.11
256	256	256	1.08	1.09
512	512	512	1.10	1.16

mMapAdd

Matrices dimensions			Ratio $(\frac{FHW}{FHW_{distilled}})$	
m1	m2	m3	writes	reads
64	64	—	1.10	1.21
128	128	—	1.07	1.14
256	256	—	1.07	1.19
512	512	—	1.10	1.21

mMapKron

Matrices dimensions			Ratio $(\frac{FHW}{FHW_{distilled}})$	
m1	m2	m3	writes	reads
2	64	—	1.71	1.88
2	128	—	1.72	1.87
2	256	—	1.65	1.83
4	128	—	1.81	1.91

Table 5: Memory accesses

Benchmark	Ratio $(\frac{FHW}{FHW_{distilled}})$								Frequency
	FDRE	LUT3	LUT6	LUT5	LUT4	LUT2	RAMB36E2	MUXF7	
mAddAdd	0.3	0.3	0.3	0.5	0.3	0.3	0.5	0.5	200 MHz
mMaskAdd	0.5	0.5	0.7	0.4	0.7	0.5	0.7	0.6	200 MHz
mMapAdd	1	0.9	0.9	1.2	1	1.1	1.1	1.2	200 MHz
mMapKron	1.5	1.5	1.3	2	2	1.8	1.4	1.7	200 MHz

Table 6: Resource utilization

6 Related Work

This section will try to show more clearly the place the work takes among adjacent works.

Firstly, the work utilizes distillation technique to provide automatic fusion of sparse linear algebra routines. At the moment, we are not aware of other works that focus on sparse linear algebra routines fusion from implementation perspective. However, it is worth noticing that SuiteSparse addresses this problem in future work [40]. The approaches that work well for dense routines fusion struggle with index arithmetic, e.g. [12], induced by sparse representation. Other approaches makes specific assumption about the operands or require to express computations with the help of combinators [38]. Our approach makes no assumptions and was shown to work successfully.

Secondly, in contrast with other works, e.g.[36], this work opts for functional high-level synthesis to produce hardware. Thus, we are not limited to only specific kernels like matrix-vector multiplication. FHW is not the only functional high-level synthesis compiler, however, it appeared to be the best fit for our pipeline. A more detailed discussion of alternatives could be found at [11]. Although we have slightly worse performance than software counterparts, the approach has a certain potential. For the same reason, we do not provide any comparison with other hardware works at the moment. Also, they are too specialized, e.g. they provide only matrix multiplication operations, while we focus on providing a reasonable GraphBLAS subset.

Finally, [29] evaluates whether supercompilation makes any benefits when a functional program is executed on a dedicated reduction-based processor *Reduceron*. Unlike this work, we use another hardware backend, distillation, and choose specific programs, namely those that contain sparse linear algebra routines. However, we plan to perform the same evaluation with Reduceron as a backend.

7 Conclusion

In the course of this work, the following results were obtained:

- Approaches for fusion in different area were studied. It was shown that distillation might be a good choice for fusion automation in the area of sparse linear algebra programs.

- Hardware generation was implemented by combining the distiller and FHW compiler. Both the distiller and the compiler were refined to have the needed functionality.

- The support for passing arguments to the `main` function in FHW was added. The passing itself works under AXI-Stream protocol and supports tree-like structures.

- A testbench was implemented by using SystemVerilog generation and TCL scripts. It allows to query execution times and collect memory usage statistics. Configurations and results of all benchmarks are stored on GitHub. The evaluation showed the success of fusion by means of distillation and slight performance degradation from hardware generation. Future work directions were outlined.

- Two posters were accepted at ICFP SRC 2021, and VPT 2022 respectively.

References

[1] ARM. AMBA®4 AXI4-Stream Protocol Specification. — 2010. — URL: https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol (online; accessed: 2022-06-07).

[2] Acharya Aravind, Bondhugula Uday, Cohen Albert. Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs // ACM Trans. Archit. Code Optim. — 2020. — sep. — Vol. 17, no. 4. — 26 p. — URL: https://doi.org/10.1145/3416510.

[3] Gupta Udit, Wu Carole-Jean, Wang Xiaodong et al. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. — 2020. — 1906.03109.

[4] Brin Sergey, Page Lawrence. The anatomy of a large-scale hypertextual Web search engine // Computer Networks and ISDN Systems. — 1998. — Vol. 30, no. 1. — P. 107 – 117. — Proceedings of the Seventh International World Wide Web Conference. URL: http://www.sciencedirect.com/science/article/pii/S016975529800110X.

[5] Cass S. Taking AI to the edge: Google's TPU now comes in a maker-friendly package // IEEE Spectrum. — 2019. — Vol. 56, no. 5. — P. 16–17.

[6] Davis T. A. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss // 2018 IEEE High Performance extreme Computing Conference (HPEC). — 2018. — P. 1–6.

[7] Davis Timothy A., Aznaveh Mohsen, Kolodziej Scott. Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse:GraphBLAS // 2019 IEEE High Performance Extreme Computing Conference (HPEC). — 2019. — P. 1–6.

[8] Davis Timothy A., Hu Yifan. The University of Florida Sparse Matrix Collection // ACM Trans. Math. Softw. — 2011. — . — Vol. 38, no. 1. — 25 p. — URL: https://doi.org/10.1145/2049662.2049663.

[9] Selvitopi Oguz, Ekanayake Saliya, Guidi Giulia et al. Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices. — 2020. — 2009.14467.

[10] Enabling massive deep neural networks with the GraphBLAS / Jeremy Kepner, Manoj Kumar, Jose Moreira et al. // 2017 IEEE High Performance Extreme Computing Conference (HPEC). — 2017. — Sep. — URL: http://dx.doi.org/10.1109/HPEC.2017.8091098.

[11] FHW Project : High-Level Hardware Synthesis from Haskell Programs / S. Edwards, Martha A. Kim, Richard Townsend et al. — 2019.

[12] Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates / Troels Henriksen, Niels G. W. Serup, Martin Elsman et al. // SIGPLAN Not. — 2017. — . — Vol. 52, no. 6. — P. 556–571. — URL: https://doi.org/10.1145/3140587.3062354.

[13] The Future is Big Graphs! A Community View on Graph Processing Systems / Sherif Sakr, Angela Bonifati, Hannes Voigt et al. // arXiv preprint arXiv:2012.06171. — 2020.

[14] The GraphBLAS C API Specification / Aydin Buluc, Timothy Mattson, Scott McMillan et al. // GraphBLAS. org, Tech. Rep. — 2017.

[15] Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines / Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams et al. // SIGPLAN Not. — 2013. — . — Vol. 48, no. 6. — P. 519–530. — URL: https://doi.org/10.1145/2499370.2462176.

[16] Hamilton Geoff. Extracting the Essence of Distillation. — 2009. — 06. — P. 151–164.

[17] Hardware Synthesis from a Recursive Functional Language / Kuangya Zhai, Richard Townsend, Lianne Lairmore et al. // Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis. — CODES '15. — IEEE Press, 2015. — P. 83–93.

[18] Johnsson Thomas. Lambda Lifting: Transforming Programs to Recursive Equations // Proc. of a Conference on Functional Programming Languages and Computer Architecture. — Berlin, Heidelberg : Springer-Verlag, 1985. — P. 190–203.

[19] Kelsey Richard A. A Correspondence between Continuation Passing Style and Static Single Assignment Form // Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations. — IR '95. — New York, NY, USA : Association for Computing Machinery, 1995. — P. 13–22. — URL: https://doi.org/10.1145/202529.202532.

[20] Kepner Jeremy, Gilbert John. Graph Algorithms in the Language of Linear Algebra. — USA : Society for Industrial and Applied Mathematics, 2011. — ISBN: 0898719909.

[21] Leskovec Jure, Sosic Rok. SNAP: A General Purpose Network Analysis and Graph Mining Library. — 2016. — 1606.07550.

[22] Linden Greg, Smith Brent, York Jeremy. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering // IEEE Internet Computing. — 2003. — . — Vol. 7, no. 1. — P. 76–80. — URL: https://doi.org/10.1109/MIC.2003.1167344.

[23] Netchain: Scale-free sub-rtt coordination / Xin Jin, Xiaozhou Li, Haoyu Zhang et al. // 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18). — 2018. — P. 35–49.

[24] Novel graph processor architecture, prototype system, and results / William S. Song, Vitaliy Gleyzer, Alexei Lomakin, Jeremy Kepner // 2016 IEEE High Performance Extreme Computing Confer-

ence (HPEC). — 2016. — Sep. — URL: http://dx.doi.org/10.1109/HPEC.2016.7761635.

[25] Pixel Visual Core: Google's Fully Programmable Image Vision and AI Processor For Mobile Devices / Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta et al. // Proc. IEEE Hot Chips Symp.(HCS). — 2018. — P. 1–18.

[26] Podlovics Peter, Hruska Csaba, Pénzes Andor. A Modern Look at GRIN, an Optimizing Functional Language Back End // Acta Cybernetica. — 2021. — 02.

[27] Predictable Accelerator Design with Time-Sensitive Affine Types / Rachit Nigam, Sachille Atapattu, Samuel Thomas et al. // Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI 2020. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 393–407. — URL: https://doi.org/10.1145/3385412.3385974.

[28] RedisGraph — GraphBLAS Enabled Graph Database / P. Cailliau, T. Davis, V. Gadepally et al. // 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2019. — P. 285–286.

[29] Reich Jason, Naylor Matthew, Runciman Colin. Supercompilation and the Reduceron. — 2010. — 12.

[30] SMASH / Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula et al. // Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. — 2019. — Oct. — URL: http://dx.doi.org/10.1145/3352460.3358286.

[31] Santos Armando, Oliveira José N. Type Your Matrices for Great Good: A Haskell Library of Typed Matrices and Applications (Functional Pearl) // Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. — Haskell 2020. — New York, NY, USA :

Association for Computing Machinery, 2020. — P. 54–66. — URL: https://doi.org/10.1145/3406088.3409019.

[32] Shaikhha Amir, Parreaux Lionel. Finally, a Polymorphic Linear Algebra Language (Pearl) // 33rd European Conference on Object-Oriented Programming (ECOOP 2019) / Ed. by Alastair F. Donaldson. — Vol. 134 of Leibniz International Proceedings in Informatics (LIPIcs). — Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. — P. 25:1–25:29. — URL: http://drops.dagstuhl.de/opus/volltexte/2019/10817.

[33] Simecek I. Sparse Matrix Computations Using the Quadtree Storage Format // 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. — 2009. — P. 168–173.

[34] SlimSell: A Vectorizable Graph Representation for Breadth-First Search / M. Besta, F. Marending, E. Solomonik, T. Hoefler // 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). — 2017. — P. 32–41.

[35] Soltaniyeh Mohammadreza, Martin Richard P., Nagarakatte Santosh. Synergistic CPU-FPGA Acceleration of Sparse Linear Algebra. — 2020. — 2004.13907.

[36] SpArch: Efficient Architecture for Sparse Matrix Multiplication / Zhekai Zhang, Hanrui Wang, Song Han, William J. Dally // 26th IEEE International Symposium on High Performance Computer Architecture (HPCA). — 2020.

[37] Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices / Xin He, Subhankar Pal, Aporva Amarnath et al. // Proceedings of the 34th ACM International Conference on Supercomputing. — ICS '20. — New York, NY, USA : Association for Computing Machinery, 2020. — 12 p. — URL: https://doi.org/10.1145/3392717.3392751.

[38] Stream Fusion, to Completeness / Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, Yannis Smaragdakis // CoRR. — 2016. — Vol. abs/1612.06668. — arXiv : 1612.06668.

[39] Sørensen Morten, Glück R., Jones Neil. A positive supercompiler // Journal of Functional Programming. — 1996. — 11. — Vol. 6. — P. 811 – 838.

[40] Timothy A.D̃avis. Algorithm 10xx: SuiteSparse:GraphBLAS: parallel graph algorithms in the language of sparse linear algebra. — 2022. — URL: https://raw.githubusercontent.com/DrTimothyAldenDavis/GraphBLAS/stable/Doc/toms_parallel_grb2.pdf (online; accessed: 2022-06-07).

[41] Timothy A. Davis Texas A&M University USA. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. — URL: https://people.engr.tamu.edu/davis/publications_files/toms_graphblas.pdf (online; accessed: 2022-06-07).

[42] Tiurin Aleksei. Quad tree matrx representation utilities. — 2022. — URL: https://github.com/Tiltedprogrammer/MatrixConverter (online; accessed: 2022-06-07).

[43] Tiurin Aleksei. Translator from `.pot` to `.grin`. — 2022. — URL: https://github.com/Tiltedprogrammer/SparseLinAlgHardware/tree/master/poitin-grin-frontend (online; accessed: 2022-06-07).

[44] Townsend Richard. Compiling Irregular Software to Specialized Hardware. — 2019.

[45] XLA: Optimizing Compiler for Machine Learning. — URL: https://www.tensorflow.org/xla?hl=en (online; accessed: 2022-06-07).

[46] Yang Carl, Buluc Aydin, Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. — 2020. — 1908.01407.