UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group High-Performance Computing

Bachelor's Thesis

Submitted to the High-Performance Computing Research Group
in Partial Fulfilment of the Requirements for the Degree of

Bachelor of Science

Accelerating Single Cell Inference of Tumor Evolution with FPGAs

by
Jan-Oliver Opdenhövel

Thesis Supervisor:
Dr. Tobias Kenter

Secondary Reviewer:
Prof. Dr. Marco Platzner

Paderborn, September 26, 2022

Abstract. SCITE is a randomized Monte Carlo Markov Chain algorithm that computes the most likely mutation history of a set of tumor cells, given noisy input data. We have accelerated SCITE using Field-Programmable Gate Arrays (FPGAs) by a factor of 8.63 relative to the reference implementation, using the Intel Stratix 10 FPGAs at the Noctua 2 supercomputer and Intel oneAPI. We also empirically verified the solution quality using the "Two One-Sided t-Tests" procedure. This performance performance has been achieved mostly by pipelining the chain step operation, which required introducing FPGA-specific algorithms to query and update data structures, restructuring the score computation function, isolating the random number generation, as well as building an optimized input/output/feedback system. The remaining bottlenecks are the score computation function due its high number of required operations and the random number generation due to a high-latency data dependency. In this thesis, we describe all our optimizations, show their correctness and how they affect the design's performance.

Contents

1 Introduction

Cancer is a widespread and often lethal disease [oDCC22] where body cells mutate in a way that increases their cell division speed as well as their lifetime, while also evading immune responses. There are many possible treatment methods like surgery, chemotherapy, or radiation therapy, but their effectiveness often depends on the exact type of tumor. Tumors however are heterogenous: Individual tumor cells may mutate again and form new subclones that compete against others [NZVLW+12]. Treating the dominant subclones however seems to provide an advantage for previously minor, resistant subclones [GVG12]. Therefore, knowledge of existing subclones and their evolutionary history may help future treatments [GM12, SCF09, Swa12].

Bulk sequencing of tumor cells and analyzing the found mutations appears to be already in wide use but seems to miss smaller, upcoming variants that are averaged out in the mass of cells [Nav14]. Therefore, research has been done to utilize single-cell sequencing. With this technique, the exact genome of individual cells can be identified and compared to other cells. However, it also comes with high error rates and parts of a cell's genome are often lost during the process [JKB16]. It is therefore very hard to identify which subclones exist and how they are related. There is therefore a need for algorithms that compute the most-likely mutation history, which is a computationally intensive task since there are many possible mutation histories to evaluate.

Such an algorithm is SCITE [JKB16], which is short for "Single Cell Inference of Tumor Evolution". The original authors of SCITE provided a functional but unoptimized implementation of their algorithm. There is however an unpublished report by Dominik Ernst et al. [EGJW20] about their efforts to optimize the algorithm for parallel CPU architectures and their list of parallelizable subtasks also contained entries that can be exploited well on FPGAs. Field-Programmable Gate Arrays (FPGAs) are computer chips that contain a lattice of logic, computation, and memory units that are connected via programmable connections. They can be used like complete, reprogrammable computer chips and have already been used extensively for chip prototyping and verification [RAMV07]. In recent years they have also become interesting as computation accelerators for High-Performance Computing (HPC) users due to their low power consumption compared to their performance [BTL10]. Our general goal for this thesis is therefore to exploit these opportunities and develop an efficient implementation of SCITE for FPGAs.

1.1 Scenario and challenges

SCITE is designed for the following scenario: A physician has been able to extract tumor tissue from a patient, either through surgery or other methods. Then, individual cells are extracted from the tissue, their genome is sequenced, and the resulting sequence of base pairs is scanned for subsequences, so-called genes. These genes may either be present in their most common form or in a less common, mutated way. The scanner identifies these mutations and produces a mutation matrix with an entry for every cell-gene combination, where 1 denotes that the cell has this mutation and 0 denotes the opposite. This matrix itself may already be interesting to evaluate, but it should be handled with caution since, as stated before, the process of amplifying and sequencing the genome is very prone to errors. While there is often only a small chance to identify a mutation where there isn't one (false positives), there is a high chance to miss a mutation (false negative). Lastly, there is also a high chance that genes are simply lost in the process. In this case, the matrix contains a third "unknown" entry encoded as 2. Jahn et al. [JKB16] quote one example with a probability for false positives of $6.04 \cdot 10^{-6}$, a probability for false negatives of 0.4309, and a probability for missing data of 0.45.

It is however possible to catch some of these errors with some assumptions on how mutations are introduced to the tumor: First of all, we assume that there are no mutations in the observed genes outside of the tumor. This means that before the tumor came into existence, there were only perfectly normal cells. When a cell replicates itself, it creates a perfect copy of its genome for one of the two new cells, and these two cells pass their genome to their subclones too. However, cells can not share or swap their genome. If a gene of a cell's genome mutates, the altered genome will therefore be passed down to its subclones and its subclones only. If we then also assume that every gene only mutates once in the whole history of the tumor (the so-called infinite sites assumption), we can arrange the mutations in their order of occurrence.

Figure 1.1 illustrates this: We assume that mutations are introduced by a single cell and are passed down to all subclones. Among these subclones, new mutations may occur, which are passed down again. The ancestry of a cell defines its genome and we can model this ancestry with a tree: In this so-called phylogenetic tree or mutation tree, every node except for the root represents a gene that may be mutated. Then, we attach every cell to a node, which expresses that this cell has all mutations on the path from the root to its attachment point, but no others. We can then construct the true mutation matrix according to this mutation tree. If we have the probabilities for false positives and false negatives given, we can then compute the likelihood that the true mutation matrix is correct.

Jahn et al. designed SCITE as a Monte Carlo Markov Chain (MCMC) algorithm to find the maximum likelihood tree. A Monte Carlo algorithm runs a random experiment that produces a possible solution to a problem and evaluates how well the generated solution solves the problem. This loop of generating and evaluating a solution is then repeated multiple times and the result is the best solution the algorithm has encountered. In theory, it would suffice if the experiment had a probability greater than zero to produce a good solution, but in order to improve the solution quality and reduce the required repetitions, one would use an experiment that produces the best solutions with a higher probability than worse solutions and that can be repeated quickly. As the name implies, MCMC algorithms are Monte Carlo algorithms that simulate a Markov Chain to produce solutions. The advantage of using Markov Chains is that the next sample may depend on the previous one and the algorithm therefore only needs to introduce small changes to the solution. This is often faster than generating a new solution from scratch and if the current sample is already a good solution, the change may preserve some of its quality. However, the designer of a MCMC algorithm has to make sure that the chain actually converges on the desired distribution.

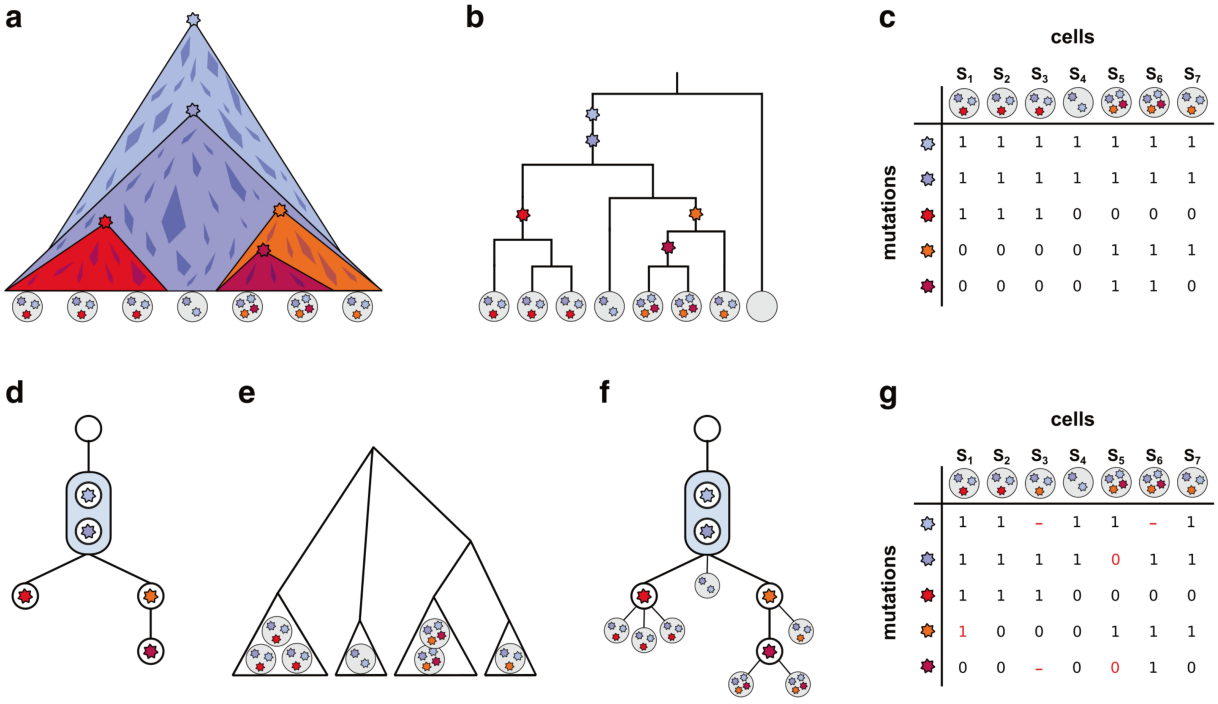


Figure 1.1: "Tumor evolution and cell phylogeny. **a** Schematic representation of tumor evolution with time progressing downwards. Stars denote new mutations leading to subclone expansion. The quadrangles belong to minor extinct subclones with no traces in the present-day populations. The mutations founding these clones may not have induced a sufficient growth advantage to have surviving descendant cells or may have been lost by chance. The gray discs on the bottom denote single cells sequenced after tumor removal. The stars they contain indicate the mutations observed in the cell. **b** Binary genealogical tree of the sequenced cells. An empty disc represents a normal somatic cell, which is an outgroup for the tumor cells. **c** Binary mutation matrix representing the mutation status of the sequenced tumor cells. A zero entry denotes the absence of a mutation in the respective cell, while a one denotes its presence. **d** The perfect phylogeny represented as a mutation tree, the partial (temporal) order of the mutation events. Mutations are summarized in a single node when their order is unidentifiable from the sampled cells, as is the case here for the two top-most mutations with the matrix from (c). **e** Hierarchical subclone structure. Cells with identical mutation profiles cluster into subclones, which serve as taxa in this phylogenetic tree. **f** Mutation tree with single-cell samples attached. **g** Noisy mutation matrix with missing values. The red numbers indicate flipped mutation states with respect to the true mutation matrix in (c). For $0 \rightarrow 1$, a false positive, the mutation is called but not present in the cell. For $1 \rightarrow 0$, a false negative, the mutation is not called but present in the cell, most likely due to allelic dropout during the DNA amplification. The red dash indicates a missing value; it is unknown whether the site is mutated or in the normal state in this cell." [JKB16], Copyright (C) Katharina Jahn et al., CC-BY 4.0, screenshot and cropped.

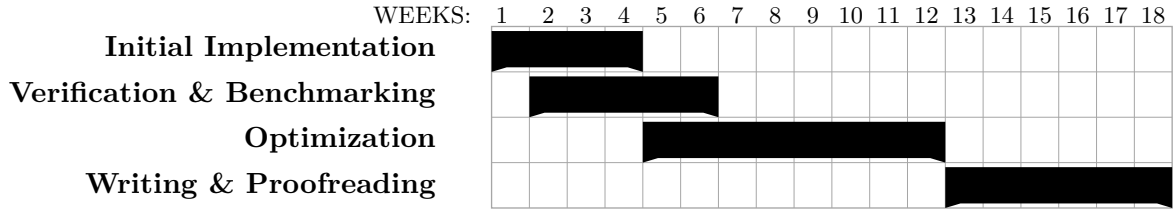


Figure 1.2: Planned worst-case work schedule

SCITE's chain state is a mutation tree and in order to transition from one chain state to the next, a random modification is proposed to this tree. This chain would however aimlessly walk through the set of all trees and would have a low probability to encounter a maximum likelihood tree. SCITE therefore only proposes a modification and computes its likelihood. If the modification improves the chain state's likelihood, it is always accepted as the new state of the chain, but if it does not, it may be rejected with a probability that rises with the decrease of likelihood produced by the modification.

There are multiple challenges in implementing the SCITE algorithm for FPGAs: First of all, we require a data structure for mutation trees that allow us to easily propose, execute and evaluate state changes. Then, there is the issue of proposing modifications: This requires the generation of random numbers, and random number generators are notorious for being hard to implement correctly and efficiently. Lastly, computing a tree's likelihood is an issue too and there are the usual problems with external memory

We faced multiple challenges during our work: First of all, we had to develop data structures and algorithms to efficiently propose, execute and evaluate state changes. Then, we had to restructure and optimize the likelihood computation method to adapt it to an FPGA's structure. We also faced issues with random number generation for tree move proposals since those implementations available to us have high resource and timing demands. Then, we had to optimize our IO and feedback structure to assure that the resulting chip design is always fully utilized. Lastly, we also had to verify that our implementation is indeed correct: As a randomized algorithm, there is a probability that even a correct implementation does not find the maximum likelihood tree. We therefore had to design a statistical test to compare our implementation with the reference.

1.2 Goals, Results, and Structure of the Thesis

Our main goals for the thesis were twofold: Firstly, we set ourselves the goal to accelerate the SCITE algorithm with FPGAs. This new implementation, which we called "fabulously fast SCITE" (ffSCITE), should perform more chain steps per second than the reference implementation by Jahn et al. [JKB16], using the Intel Stratix 10 GX 2800 FPGAs and the AMD Milan 7763 Central Processing Units (CPUs) of the Noctua 2 supercomputer at the Paderborn University, respectively. Secondly, we set the goal to design and apply a statistical set to verify that SCITE and ffSCITE produce solutions of equivalent quality. Additionally, we set ourselves the optional goals to achieve higher throughput than the optimized CPU implementation by Ernst et al. [EGJW20] and to adapt our implementation to the scoring model of Infinity-SCITE (∞SCITE) published by Kuipers et al. [KJRB17], which is an improved version of SCITE. We planned to work with the worst-case schedule outlined in figure 1.2.

The final build of ffSCITE achieves a constant throughput of 566.72 thousand chain steps per second, which is a speed up of up to 8.63 compared to SCITE. This matches our performance

model which predicted that the design requires 512 cycles to process one step at the clock speed of 295.83 MHz. Our quality test is based on the "Two One-Sided t-Tests" (TOST) procedure [Sch87] and we were able to show that ffSCITE is equivalent to SCITE with a significance level of 2%. However, we assume that the implementation by Ernst et al. [EGJW20] has higher throughput than ffSCITE and we were not able to evaluate the extensibility of ffSCITE to ∞SCITE. We also had to remove some features of SCITE in ffSCITE in order to reach our goal in time, for example collecting all co-optimal trees and the β search.

The rest of this thesis is structured as follows: First, we provide a detailed and formal overview of SCITE and fundamental concepts of FPGA designs in chapter 2. In the following chapter 3, we assume that readers are familiar with the concepts introduced in chapter 2 and explain the design decisions that lead to the final design of ffSCITE. Lastly, we evaluate the different properties of ffSCITE in chapter 4, namely the hardware resource usage, the throughput benchmark, and the quality test, and summarize all contributions in chapter 5. This chapter also contains a list of smaller, optional features that we did not implement and open directions for further optimization and analysis.

2 Background

In this chapter, we provide background information to give readers the required knowledge to follow the thesis. First, we will fully introduce the SCITE algorithm and the formalisms to describe it, and second, we give an overview on how FPGAs work and how efficient programs for them are structured.

2.1 Introduction to SCITE

In this section, we describe the theoretical framework in which this thesis operates. We will first introduce the observed phenomenon of mutations and the erroneous mutation matrix we receive from single-cell sequencing. Then, we will introduce the likelihood function we wish to maximize, and the mutation trees and attachment functions that we use to model the true mutation status of the cells. Lastly, we introduce the SCITE Markov chain which uses all of the previous concepts to find a maximum-likelihood mutation tree. All of this is conceptually equivalent to the framework used by Jahn et al. [JKB16], but with enhanced precision and details. Equivalent formalizations are found in either the original paper or implementation if not indicated otherwise. First, we will formalize the observed phenomenon of mutations:

Definition 2.1 (Mutation matrix)**.** Let C be the set of cells with $|C| \in \mathbb{N}$ and G be the set of genes with $|G| \in \mathbb{N}$. Then, we define the random variables E and D with $\text{im}(E) \subseteq \{0,1\}^{|C|\times|G|}$ and $\text{im}(D) \subseteq \{0,1,2\}^{|C|\times|G|}$. E is the true mutation matrix that describes the actual state of the cells, and D is the observed, erroneous mutation matrix. Let $\alpha \in (0,1)$ be the probability of false positives and $\beta \in (0,1)$ be the probability of false negatives. We then assume the following for all $c \in C, g \in G$:

$$\mathbb{P}(D_{c,g} = 1 \mid E_{c,g} = 0 \wedge D_{c,g} \neq 2) := \alpha \qquad \mathbb{P}(D_{0,j} = 0 \mid E_{c,g} = 1 \wedge D_{c,g} \neq 2) := \beta$$
$$\mathbb{P}(D_{c,g} = 0 \mid E_{c,g} = 0 \wedge D_{c,g} \neq 2) := (1-\alpha) \qquad \mathbb{P}(D_{c,g} = 1 \mid E_{c,g} = 1 \wedge D_{c,g} \neq 2) := (1-\beta)$$

The additional prior $D_{c,g} \neq 2$ is not present in the original formalization, but technically necessary since we would otherwise have $\mathbb{P}(D_{c,g} = 2 \mid E_{c,g} = e) = 0$ for all $e \in \{0,1\}$, which is not useful.

Although this thesis was initially very applied in nature, it has become very theoretical. Therefore, we have decided to use theoretical indexing conventions in this thesis. Therefore, we use the first index to indicate the column of the matrix and the second index to indicate the row. We also use 1 as our first index and upper index bounds are inclusive.

Definition 2.2 (Likelihood function)**.** We define the elementary likelihood function λ as follows:

$$\lambda : \{0,1,2\} \times \{0,1\} \to [0,1], (d,e) \mapsto \begin{cases} 1-\alpha & d=0 \wedge e=0 \\ \alpha & d=1 \wedge e=0 \\ \beta & d=0 \wedge e=1 \\ 1-\beta & d=1 \wedge e=1 \\ 1 & d=2 \end{cases}$$

We, therefore, have

$$\lambda(d_{c,g}, e_{c,g}) = \mathbb{P}(D_{c,g} = d_{c,g} \mid E_{c,g} = e_{c,g} \wedge D_{c,g} \neq 2)$$

for all $d \in \{0,1,2\}^{n \times m}, e \in \{0,1\}^{n \times m}, g \in G, c \in C$ with $d_{c,g} \neq 2$. We make the distinction between $d_{c,g} = 2$ and $d_{c,g} \neq 2$ since we are not interested in lost data, with the underlying assumption that data loss is independent of the true mutation status of a cell. This distinction is not present in the original formalization, but it is present in the original implementation. Given an observed mutation data matrix $d \in \{0,1,2\}^{n \times m}$, we then define the global likelihood function as follows:

$$\Lambda_d : \{0,1\}^{|C| \times |G|} \to [0,1], e \mapsto \prod_{c \in C} \prod_{g \in C} \lambda(d_{c,g}, e_{c,g})$$

This function Λ_d is the likelihood function we wish to maximize. We could now define the problem using only mutation matrices, but since Jahn et al. make no statement whether SCITE also finds the maximum-likelihood mutation matrix, we restrict ourselves to maximum-likelihood mutation trees, which we will define now:

Definition 2.3 (Mutation trees)**.** Let G be the set of considered genes. A mutation tree is a rooted, directed tree $T = (V, E, r)$ with $V = G \cup \{r\}$. We, therefore, have $E \subseteq V^2$ and $|V| = |G| + 1$.

Definition 2.4 (Tree-related notations)**.** Let $T = (V, E)$ be a tree and $v, w \in V$. w is reachable from v, formally $v \rightsquigarrow_T w$, iff there is a path $p = (v, \ldots, w)$ in T. We also define the parent of v, formally $p_T(v) \in V$, as the node with $(p_T(v), v) \in E$ if such a node exists. These notations are generally well known and do not necessarily need an introduction, but we want to assure that the reader is aware of them.

Definition 2.5 (Attachment functions)**.** Let C and G be the set of considered cells and genes, respectively, and T a mutation tree. An attachment function is a function $\sigma : C \to V$ that maps cells onto tree nodes.

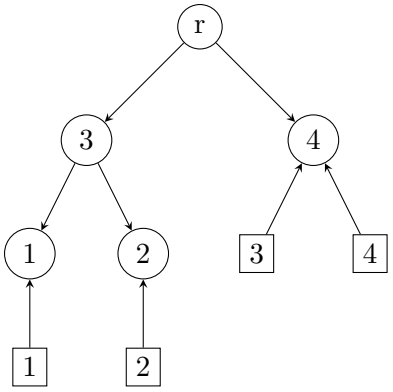
Definition 2.6 (Induced mutation matrix)**.** Let C and G be the set of considered cells and genes, respectively, T be a mutation tree and σ be an attachment function. We define the mutation matrix $e_{T,\sigma} \in \{0,1\}^{|C| \times |G|}$ induced by the mutation matrix and attachment function as follows:

$$\forall c \leq C, g \leq G : (e_{T,\sigma})_{c,g} := \begin{cases} 1 & g \rightsquigarrow_T \sigma(c) \\ 0 & \text{else} \end{cases}$$

This means that a cell c has a mutation at the gene g iff g is an ancestor of its attachment node $\sigma(c)$. An example of an induced mutation matrix is found in example 2.8.

Definition 2.7 (The SCITE problem)**.** Let C and G be the set of considered cells and genes, respectively, and $d \in \{0,1\}^{|C|\times|G|}$ an observed mutation matrix. The SCITE problem is: Find a mutation tree T and an attachment function σ so that $\Lambda_d(e_{T,\sigma})$ is maximal.

Example 2.8. We consider 4 cells and 4 genes in this example and therefore set $C = \{1,2,3,4\}$ and $G = \{1,2,3,4\}$. We assume that the true, underlying mutation tree T and attachments σ look like this:



In this graph, the encircled nodes represent the actual nodes of the mutation tree, while the squared nodes represent cells that are attached to a node. Therefore, this gives us the following true, induced mutation matrix:

$$e_{T,\sigma} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

For example, cell number 1 is attached to node number 1, which means that we have $\sigma(1) = 1$. Since only nodes 1 and 3 are on the path from the root to the attachment point $\sigma(1)$, there are 1-entries exactly at positions $(1,1)$ and $(3,1)$ of $e_{T,\sigma}$, but not at $(2,1)$ and $(4,1)$.

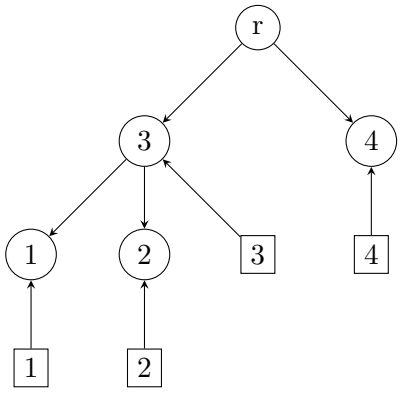
We now assume that $(\alpha,\beta) = (0.01, 0.4)$ and that matrix entries are reported as missing with a probability of 0.25, after the false-positive and false-negative errors are decided. This is not necessarily the case for all inputs or the real world, but we can use this model to generate an observed mutation matrix d, like this one:

$$d = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, we can compute the likelihood of the underlying mutation tree and attachment:

$$\begin{aligned} \Lambda_d(e_{T,\sigma}) = (1-\alpha)^8 \cdot \alpha^0 \cdot (1-\beta)^4 \cdot \beta^1 &\approx 0.05 \\ &\approx \exp(-3.04) \end{aligned}$$

However, if you attach cell 3 to gene 3 instead, you get the following tree:



with the following true mutation matrix:

$$e_{T,\sigma'} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and the following likelihood:

$$\begin{aligned} \Lambda_d(e_{T,\sigma'}) = (1-\alpha)^9 \cdot \alpha^0 \cdot (1-\beta)^4 \cdot \beta^0 &\approx 0.12 \\ &\approx \exp(-2.13) \end{aligned}$$

This is one of the solutions found by SCITE. This shows that the true, underlying mutation tree and attachment function is not necessarily the most likely one. It is not easy to tell whether this is a problem with the model itself or whether this information is simply lost in the errors. However, we have decided to not look into this issue as it does not align with our goals.

Next, we have a lemma that shows that it is not necessary to search for the maximal attachment function given a mutation tree:

Lemma 2.9 (Maximum-likelihood attachment function)**.** *Let C and G be the set of considered cells and genes, respectively, $d \in \{0,1\}^{|C|\times|G|}$ an observed mutation matrix, and T a mutation tree. We define:*

$$\sigma_{\max,T} : C \to V, c \mapsto \arg\max_{v \in V} \prod_{g \in G} \lambda \left(d_{c,g}, \begin{cases} 1 & g \rightsquigarrow_T v \\ 0 & \text{else} \end{cases} \right)$$

$\sigma_{\max,T}$ maximizes $\Lambda_d(e_{T,\sigma})$ among all $\sigma : C \to V$.

Informally, $\sigma_{\max,T}$ simply picks the most-likely attachment node for every individual cell and using this attachment function yields the maximum likelihood for every tree, respectively. This lemma is already assumed to be true in the original SCITE paper [JKB16]. Therefore, we see no need to prove it. However, it means that we only need to try different trees, not tree-attachment combinations, and we can simply write $\Lambda_d(T)$ instead of $\Lambda_d(e_{T,\sigma_{\max,T}})$ to express the likelihood of a mutation tree. Now, we can finally describe the SCITE Markov chain and algorithm:

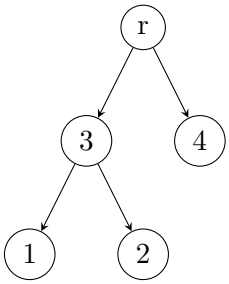
Definition 2.10 (SCITE Markov chain, [JKB16])**.** We define the SCITE Markov chain as the Markov chain $(T_i)_{i\in\mathbb{N}_0} \in \{(V,E,r) : (V,E,r) \text{ is mutation tree}\}$. T_0 is uniformly distributed over the set of all mutation trees, and for every $i \in \mathbb{N}_0$, we set $T_{i+1} := \textsc{ChainStep}(T_i)$.

The ChainStep algorithm (Algorithm 2.1) works by introducing one of three modifications to the tree and accepting it as the new state with a probability related to the increase in likelihood. The first of these modifications is the "swap nodes" move. In this move, two distinct non-root nodes v and w are sampled and their position in the tree is swapped. The resulting edge set is just defined as the original edge set where every occurrence of v and w are swapped. "swap nodes" is therefore also referred to as "label swap" since if the node-gene relationship were modeled with a node labeling function, one would simply need to swap the labels of v and w. The next and most basic modification is "prune and reattach." In this move, a node v is sampled from all non-root nodes and a node w is sampled from all nodes that are not descendants of v, including the root r. Then, the node v is simply attached to w. Lastly, there is the "swap subtrees" modification. Here, two distinct non-root nodes v and w are sampled too. If v and w are unrelated (formally $v \not\rightsquigarrow_T w \wedge w \not\rightsquigarrow_T v$), v is simply attached to w's former parent and w is attached to v's former parent. However, if we have $w \rightsquigarrow_T v$, a third node t is sampled from all descendants of v and w is attached to t instead of v's parent. This assures that no circles are introduced to the tree. If we have $v \rightsquigarrow_T w$, we can simply swap v and w and do the same.

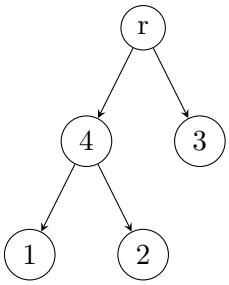
If we would now accept every modified tree, we would eventually encounter a high- or even maximum-likelihood tree. However, the probability of this is low since the chain wanders through the search space aimlessly. Therefore, Jahn et al. employed a rejection sampling method that accepts solutions with a high likelihood and rejects solutions with a low likelihood. The exact way how this is achieved is documented in algorithm 2.1 and according to the authors, this assures that the chain converges on trees with a high likelihood.

The SCITE algorithm simulates the SCITE Markov chain for a user-requested number of steps and repetitions and outputs the state T with the highest $\Lambda_d(T)$ it has encountered. The goal of the thesis is to simulate the SCITE Markov chain as quickly as possible, at least as quickly as the original implementation.

Example 2.11. We will give examples of different chain steps. We use the same tree as in example 2.8, but without the attached cells since these are not relevant now:



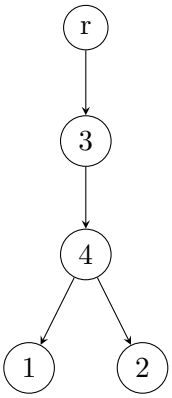
If we now execute a "swap nodes" move with $v = 3$ and $w = 4$, the resulting tree looks like this:



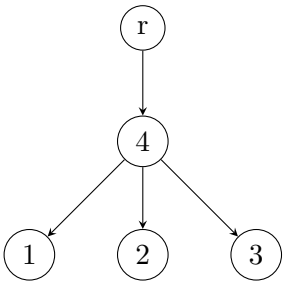
Nodes 4 and 3 just have swapped their labels. Everything else is still the same. Now, we execute the "prune and reattach" move with $v = 4$ and $w = 3$, so we take node 4 and attach it to node 3. This is possible since there is no path from node 4 to node 3:

function ChainStep(T)
  $(V, E, r) \leftarrow T$
  move $\leftarrow$ sample uniformly from {"swap nodes", "prune and reattach", "swap subtrees"}
  if move = "swap nodes" **then**
    $v \leftarrow$ sample uniformly from $V \setminus \{r\}$
    $w \leftarrow$ sample uniformly from $V \setminus \{r, v\}$
    $f : V \to V, x \mapsto \begin{cases} w & x = v \\ v & x = w \\ x & \text{else} \end{cases}$
    $E' \leftarrow \{(f(x), f(y)) : (x, y) \in E\}$
    $f_{\text{correction}} \leftarrow 1$
  else if move = "prune and reattach" **then**
    $v \leftarrow$ sample uniformly from $V \setminus \{r\}$
    $w \leftarrow$ sample uniformly from $\{w \in V : v \not\rightsquigarrow_T w\}$
    $E' \leftarrow (E \setminus \{(p_T(v), v)\}) \cup \{(w, v)\}$ ▷ Remove the old edge, add the new one
    $f_{\text{correction}} \leftarrow 1$
  else if move = "swap subtrees" **then**
    $v \leftarrow$ sample uniformly from $V \setminus \{r\}$
    $w \leftarrow$ sample uniformly from $V \setminus \{r, v\}$
    if $v \rightsquigarrow_T w$ **then**
      $v, w \leftarrow w, v$ ▷ Ensure that v is always a descendant or unrelated to w.
    end if
    if $w \rightsquigarrow_T v$ **then**
      $t \leftarrow$ sample uniformly from $\{x \in V : v \rightsquigarrow_T x\}$
      $f_{\text{correction}} \leftarrow \frac{|\{x \in V : v \rightsquigarrow_T x\}|}{|\{x \in V : w \rightsquigarrow_T x\}|}$
    else
      $t \leftarrow p(v)$
      $f_{\text{correction}} \leftarrow 1$
    end if
    $E' \leftarrow (E \setminus \{(p_T(v), v)\}) \cup \{(p_T(w), v)\}$ ▷ Remove the old edge to v, add the new one
    $E' \leftarrow (E' \setminus \{(p_T(w), w)\}) \cup \{(t, w)\}$ ▷ The same as above for w
  end if
  $T' \leftarrow (V, E', r)$
  if accept with probability $\left(f_{\text{correction}} \cdot \frac{\Lambda_d(T')}{\Lambda_d(T)}\right)$ **then**
    return T'
  else
    return T
  end if
end function

Algorithm 2.1: Algorithm to compute the next chain step of a SCITE Markov Chain, adapted from [JKB16]



Now, we will give an example of the "swap nodes" move with related subtrees. Firstly, we choose $v = 4$ and $w = 3$. Since we have $w \rightsquigarrow_T v$, we don't need to swap v and w, but we need to sample a new target node t among the descendants of v. Since every node is also its own descendant, we can choose $t = v = 4$. Now, we attach $v = 4$ to r, which is the parent of $w = 3$, and attach 3 to $t = 4$. The resulting tree looks like this:



2.2 FPGA fundamentals

Our thesis requires a certain level of knowledge regarding FPGAs and their characteristics. Since FPGAs are still a rather niche platform and common patterns deviate significantly from common programming practices, we will introduce some concepts of FPGAs and how they are used. An interested reader may however find more details in Intel's "FPGA Optimization Guide for Intel oneAPI Toolkits." and the book "Parallel Programming for FPGAs" by Kastner, Matai, and Neuendorffer [KMN18].

Field-Programmable Gate Arrays (FPGAs) are programmable computer chips, but not in the sense of CPUs which execute a list of instructions read from external memory. Instead, FPGAs are a lattice of small, reprogrammable blocks with reprogrammable connections. There are logic blocks, which contain a look-up table (LUT) that maps every bit combination to another, arbitrary bit combination, and a flip-flop (FF), which can store a value for one clock cycle. There are also random access memory (RAM) blocks with can store multiple values over many clock cycles and digital signal processor (DSP) blocks with purpose-built floating-point operation circuits. With these blocks, any other digital circuit can be constructed and to other computer chips, a programmed FPGA acts just like the programmed chip design. FPGAs are therefore often used to simulate and test planned chip designs that are later implemented in silicon, or to replace chips that are unprofitable to manufacture.

In a high-performance computing context, however, FPGAs fulfill a different role: Here, they are used as computation accelerators and are therefore found on PCIe extension cards inside supercomputer clusters. Another difference is that computational designs are usually written in common, high-level languages like C or C++ instead of hardware description languages like

VHDL or Verilog since developers of FPGA-accelerated applications are usually application developers and software engineers, not electrical engineers. In our case, we used Intel's oneAPI and implemented the entire design and the accompanying host application in one C++ codebase. The `dpcpp` compiler extracts the part of the code that is supposed to be executed by the FPGA and lowers it to an equivalent hardware design.

Most of the code that can run on FPGAs can also run on CPUs, and with some exceptions also vice versa, due to this toolchain. However, one needs to change their mental model on how code is executed in order to write efficient FPGA code. In the following, we will describe a mental model to work with FPGAs. We however want to emphasize that we are not electrical engineers and only work with FPGAs on a high abstraction level, so some details may or may not be entirely correct. Let's take algorithm 2.2 as an example. This algorithm receives the real and imaginary parts of a sequence of complex numbers, computes their absolute value, and sums them up to return the result. A normal, single-threaded CPU would walk through the algorithm step by step and first load a_i and b_i from memory, square both values, sum the result, take the square root of the result, add it to the current value of s and store the result in s. It is a linear sequence of operations and the CPU is always occupied with loading the instruction from memory, decoding it, executing it, and writing the results back. On FPGAs, every instruction is a piece of hardware, a small area on the chip. There will be a dedicated combination of different blocks to load a_i and b_i, as well as for every square, sum, and square root operation. These areas are connected and can be thought of as receiving their input from the previous area and sending the result to the next; Figure 2.1 illustrates this.

function AbsoluteSum(n, $(a_i)_{i\leq n}$, $(b_i)_{i\leq n}$)
  $s \leftarrow 0$
  for all $i \in \{1, 2, \ldots, n\}$ **do**
    $s \leftarrow s + \sqrt{a_i^2 + b_i^2}$
  end for
  return s
end function

Algorithm 2.2: Example algorithm that sums up the absolute value of complex numbers, where the real parts are stored in a and the imaginary parts are stored in b.

The implications of this are that the runtime behavior is very different on FPGAs and on CPUs. For example, a normal CPU would execute the two square operations serially, but on an FPGA they are executed in parallel, in two separate blocks. The fact that operations can be executed independently in different areas of the chip is the source of one of the most important techniques on FPGAs: Pipelining. If we insert registers on the edges between instruction nodes, earlier stages of the pipeline can start to work on the next iteration while the following stages finish previous iterations. In our example, we can let the reading instructions read one number per cycle and pass the result down step by step. If it were implemented like this, the loop would have an initiation interval (II) of 1 cycle, a latency of 5 cycles, and a capacity of 5 iterations. This means that it can initiate a new iteration every cycle, that it takes five cycles until the result of one iteration has reached the end of the pipeline, and that five iterations can be "on the fly" at once.

The goal during FPGA design development is to find a way to implement the loops of an algorithm with a minimal II, since this leads to a maximum amount of iterations that are "on the fly" and therefore throughput. One of the biggest hazards for this is feedback: Information that is computed during one iteration of the loop and required for one or more following iterations. In the worst case, early instructions need to wait for a specific number of cycles until the information

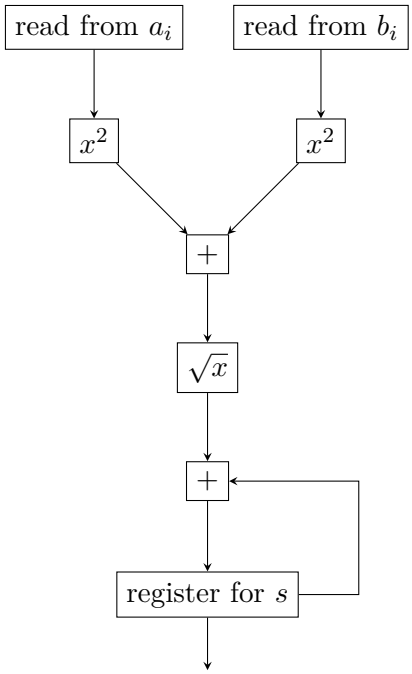


Figure 2.1: Schematic of how the loop body of algorithm 2.2 may be implemented on FPGAs.

they require is available. This will increase the II of the loop and therefore the throughput. In our example, there is feedback on the variable s, and if we were to place additional registers between the "+" and "register for s" nodes, "+" would need to wait for one additional cycle until the resulting value is available and the II would be 2. However, we can feed the result of "+" directly back and resolve this issue. Finding such solutions is the task of the compiler, not the developer, but the compiler often needs a special wording of the program in order to work correctly. Finding the best wording that leads to the best design is one of the more tedious tasks of FPGA programming.

Another source of performance on FPGAs is loop unrolling. With this technique, the compiler is instructed to replicate the entire loop body, so that even more iterations can be processed in parallel. However, the compiler may need to insert additional, complicated logic to combine the results of two loop body replicas, which may worsen the loop characteristics again. Loop unrolling also increases hardware usage; A developer therefore needs to do a design space exploration to find the optimal amount of unrolling for their specific loop.

Memory management is also another big strength of FPGAs, but also a challenge for the developer: On a modern computer, the CPU has multiple levels of cache: Small and fast memory blocks near the cores. A program only requests to access a certain position in the global external memory and the CPU tries to predict which areas of the memory a program is going to access and loads them in the cache. Programs therefore need to be written in a way that is easy to predict by the CPU. For a FPGA however, there is no pre-built cache, only external memory. The application developer therefore needs to build their own caching structure for their program to optimize the memory access. This is of course a big strength of FPGAs, since application developers may exactly know which data their algorithm needs and can therefore design their access patterns without needing to dynamically predict the accesses. However, this becomes increasingly complicated with complex memory access patterns.

Lastly, implementing an entire algorithm in one piece often brings the advantage that the compiler can analyze the entire algorithm and find shortcuts and optimizations that the developer might not expect. However, too complicated code may also "confuse" the compiler and

may lead to not matching or failing heuristics, or it may simply not be possible to describe the desired physical design as one sequence of instructions. In this case, the algorithm can be broken up into so-called kernels. These kernels are described in separate pieces of code and one can think of them as internal chips or as different dies in a multi-die package. Kernels can also be connected using pipes. Pipes are inter-kernel connections where one kernel can write data into the pipe and a different kernel can read data from the pipe in a first in-first out fashion. This allows kernels to communicate with one another without using external memory.

3 Design decisions

Our implementation of SCITE is a multi-kernel design. We initially tried to implement the algorithm with a single kernel, but we soon encountered feedback problems that we were not able to resolve inside a single kernel. Therefore, we split the parts of the algorithm with problematic feedback into separate kernels, connected via pipes. Figure 3.1 gives an overview of the design: The change proposer kernel receives the current state from the IO kernel and samples all random parameters of a chain step. Everything else like finding the deterministic parameters of the move, computing the resulting tree, computing its likelihood, and evaluating it is done by the "Tree Scorer" kernel. It contains four internal loops that have to be completed for every tree. Once the likelihood score of the new tree is computed and the new current state is decided, it is fed back to the IO kernel. Additionally, it keeps track of the most-likely state it encounters and writes this state to a global buffer once the execution is finished. The change proposer kernel and the loops inside the tree scorer kernel all operate independently of each other in a macro-pipeline. The IO kernel dispatches and receives states for the pipeline: Chain states from the tree scorer are fed back to the pipeline as often as requested by the user and when these chains are finished, new initial states are read from global memory. These states are precomputed by the host since we did not want to waste chip space on hardware that is rarely used. It is also not necessary to write the intermediate or final chain states back to global memory since the application only evaluates the best states.

In this chapter, we discuss noteworthy contributions and optimizations. First, we describe how we used both logical properties of mutation trees and technical properties of FPGAs to efficiently use, modify and transfer mutation trees in the design. Secondly, we describe how we have rewritten the likelihood score computation to make use of wide parallelism available on FPGAs. Thirdly, we describe our difficulties with move proposals, especially with random number generators, and how we have minimized their impact. Lastly, we describe how we have set up the IO and feedback kernel to saturate the computation pipeline.

3.1 Tree encoding and operations

Our first and most impactful contribution is an improvement to the used mutation tree encoding and the operations on this code. The original SCITE implementation [JKB16] uses a parent vector as the canonical data structure to encode a mutation tree:

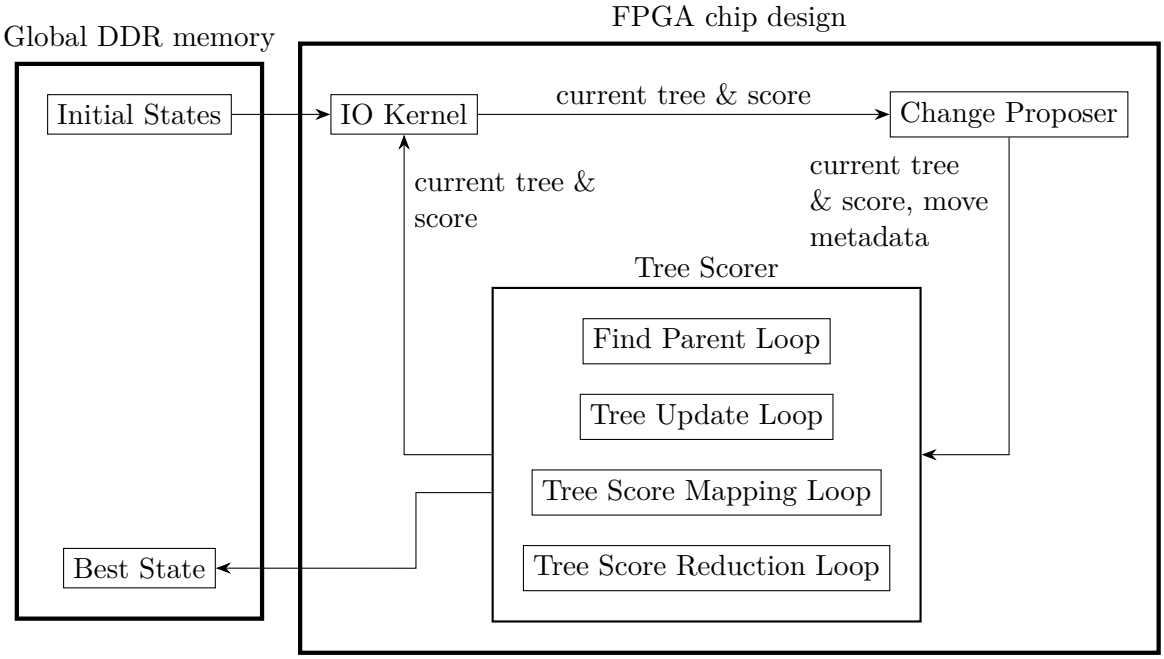


Figure 3.1: Schematic of the ffSCITE chip design.

Definition 3.1 (Parent vector, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree. The corresponding parent vector is defined as the sequence $(p_v)_{v \in V} \subseteq V$ with

$$p_v := \begin{cases} p_T(v) & v \neq r \\ r & v = r \end{cases}$$

Using this encoding has the obvious advantage that most of the tree moves are simple: The updates in the "prune and reattach" and "swap subtrees" moves of ChainStep (Algorithm 2.1) are mere constant operations. The update operations in the "swap nodes" move are more involved since every edge needs to be visited and checked, but it is still in linear runtime. However, sampling moves and computing the likelihood function requires many connectivity queries: Both the "prune and reattach" and the "swap subtrees" moves may need to sample a target that is not a descendant of the moved node, and the induced mutation matrix (Definition 2.6) is defined by node connectivity. Therefore, a separate data structure is used to answer these queries quickly:

Definition 3.2 (Ancestor matrix, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree. The corresponding ancestor matrix is defined as the matrix $A \in \{0, 1\}^{|V| \times |V|}$ with

$$A_{v,w} := \begin{cases} 1 & v \rightsquigarrow_T w \\ 0 & \text{else} \end{cases}$$

for all $v, w \in V$.

Jahn et al. [JKB16] also give an algorithm that constructs an ancestor matrix from a parent vector; We have listed it as algorithm 3.1. Intuitively, it walks up from every node to the root and marks all nodes it encounters as ancestors. SCITE uses this algorithm once per chain step to sample descendants and non-descendants for a move. Hardware implementations of this algorithm are however inefficient since it is hard to predict how often the inner while-loop is

executed. For example, it may be executed $|V|$ times for the leaf of a completely degenerated tree, but it may not be executed at all for the root r. Therefore, the outer for-loop needs to be executed serially, which severely limits the performance of the design. However, we were able to eliminate the need to construct ancestor matrices on the device. First of all, we were able to show with algorithm 3.2 that it is possible to find a node's parent using an ancestor matrix. Ancestor matrices can therefore be used as the canonical data structure to encode mutation trees. More importantly, however, we were able to show with algorithms 3.3, 3.4, 3.5 and 3.6 that every move of the ChainStep algorithm (Algorithm 2.1) can be executed on an ancestor matrix with linear time and space requirements. This leaves us with one linear, perfectly pipelinable loop to update the ancestor matrix for a move. Compared to a quadratic, barely pipelinable loop to compute the ancestor matrix twice, this is certainly an improvement.

```
1: function AncestorMatrix((p_v)_{v∈V} ⊆ V, r ∈ V)        ▷ r is the root of the tree.
2:     n ← |V|
3:     A ← 0 ∈ {0,1}^{n×n}
4:     for all w ∈ V do
5:         v ← w
6:         while v ≠ r do
7:             A_{v,w} ← 1
8:             v ← p_v
9:         end while
10:        A_{r,w} ← r
11:    end for
12:    return A
13: end function
```

Algorithm 3.1: Algorithm to construct an ancestor matrix (Definition 3.2) from a parent vector (Definition 3.1), [JKB16]

The remainder of this section is structured as follows: We first will introduce the general algorithms mentioned above and argue why they are correct. The first algorithm to be discussed is IsParent, which evaluates whether one node is the other node's parent, and the following algorithms are used to compute the resulting ancestor matrix of a tree move, given the previous ancestor matrix. These subsections first introduce the move again formally, present the algorithm, provide an example and argue for their correctness. Then, we describe how we implemented those algorithms and how we achieved the runtime and space behavior mentioned above.

3.1.1 Reversing the ancestor matrix construction

We introduce the algorithm IsParent (Algorithm 3.2) to answer the query whether one node is another node's parent. It simply evaluates whether the right-hand side of the following lemma 3.3 is true and returns the result. However, it also catches the case where $v = w = r$ since we used the convention that the root is the one node that is its own parent. Since the for-loop of IsParent is unrolled and therefore has a runtime in $O(1)$, one can use IsParent in a loop to find a node's parent in $O(|V|)$.

Lemma 3.3. *Let $T = (V, E)$ be a tree and $v, w \in V$. We have:*

$$(v, w) \in E \Leftrightarrow (\forall x \in V \setminus \{w\} : x \rightsquigarrow_T v \Leftrightarrow x \rightsquigarrow_T w) \wedge (v \rightsquigarrow_T w)$$

```
1: function IsParent(V, A ∈ {0,1}^{|V|×|V|}, (v, w) ∈ V²)
2:     if A_{v,w} = 0 then
3:         return False
4:     end if
5:     if v = w then
6:         return v = r        ▷ Per convention, the root is the only node that is also its parent.
7:     end if
8:     for all x ∈ V \ {w} do                                   ▷ Unroll completely
9:         if A_{x,w} ≠ A_{x,v} then
10:            return False
11:        end if
12:    end for
13:    return True
14: end function
```

Algorithm 3.2: Algorithm to query whether an edge exists in a tree, using an ancestor matrix

Proof. We first show $\Rightarrow$: We obviously have $v \rightsquigarrow_T w$. Let $x \in V \setminus \{v\}$. Then, we have:

$$\begin{aligned} x \rightsquigarrow_T w &\Rightarrow \exists p = (x, \dots, w) \subseteq E \\ &\overset{(v,w)\in E}{\Rightarrow} (v, w) \in p \\ &\Rightarrow p' := p \setminus \{(v, w)\} = (x, \dots, v) \subseteq E \\ &\Rightarrow x \rightsquigarrow_T v \\ x \rightsquigarrow_T v &\Rightarrow \exists p = (x, \dots, v) \subseteq E \\ &\overset{(v,w)\in E}{\Rightarrow} p' := p \cup \{(v, w)\} = (x, \dots, v, w) \subseteq E \\ &\Rightarrow x \rightsquigarrow_T w \end{aligned}$$

Now, we show $\Leftarrow$: Let's assume for a contradiction that we have $(\forall x \in V \setminus \{w\} : x \rightsquigarrow_T v \Leftrightarrow x \rightsquigarrow_T w) \wedge (v \rightsquigarrow_T w)$ and $(v, w) \notin E$. We have:

$$\begin{aligned} v \rightsquigarrow_T w \wedge (v, w) \notin E &\Rightarrow \exists y \in V \setminus \{v, w\} : v \rightsquigarrow_T y \rightsquigarrow_T w \\ y \rightsquigarrow_T w \wedge y \neq w &\Rightarrow y \rightsquigarrow_T v \\ &\Rightarrow y \rightsquigarrow_T v \rightsquigarrow_T y \end{aligned}$$

This means that our tree has a circle, which is a contradiction. Therefore, such a y can not exist and we have in fact $p = (v, \dots, w) = (v, w) \Rightarrow (v, w) \in E$. □

3.1.2 "Swap nodes" move

The first move we discuss is the "swap nodes" move:

Definition 3.4 (Swap nodes move, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree, and $v, w \in V \setminus \{r\}$ with $v \neq w$. We define the mutation tree T' after the "swap nodes" move as $T' = (V, E', r)$ with

$$E' := \{(f(x), f(y)) : (x, y) \in E\}$$

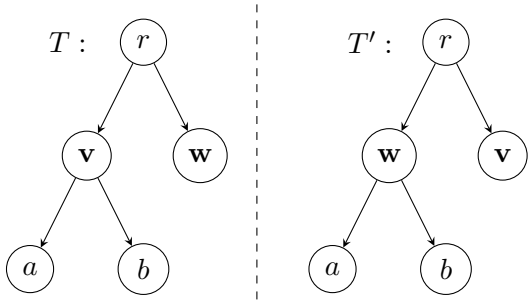


Figure 3.2: Example tree to explain the "swap nodes" move.

where f is defined as

$$f : V \to V, x \mapsto \begin{cases} v & x = w \\ w & x = v \\ x & \text{else} \end{cases}$$

In essence, the move swaps the labels of the nodes that were previously labeled as v and w, as indicated in figure 3.2. We present the algorithm AM::SwapNodes 3.3 that computes the resulting ancestor matrix of the "swap nodes" move.

```
1: function AM::SwapNodes(V, A ∈ {0,1}^{|V|×|V|}, v, w)
2:     A′ ← 0 ∈ {0,1}^{|V|×|V|}
3:     for all x ∈ V do
4:         if x = v then
5:             A′[x] ← A[w]
6:         else if x = w then
7:             A′[x] ← A[v]
8:         else
9:             A′[x] ← A[x]
10:        end if
11:        A′[x][v], A′[x][w] ← A′[x][w], A′[x][v]                ▷ Bit Swap
12:    end for
13:    return A′
14: end function
```

Algorithm 3.3: Algorithm to perform the "swap nodes" move on an ancestor matrix. All edges from and to v are w are swapped, assuming that we have $v \neq w$.

The underlying principle of the algorithm is expressed in the following lemma, which says that two nodes x and y are connected in T' iff $f(x)$ and $f(y)$ are connected in T:

Lemma 3.5. *We have $x \rightsquigarrow_{T'} y \Leftrightarrow f(x) \rightsquigarrow_T f(y)$ for all $x, y \in V$.*

Proof. First, it should be noted that f is obviously self-inverse, so that we have $f(f(x)) = x$ and $f(f(T)) = T$. We therefore only need to show $x \rightsquigarrow_T y \Rightarrow f(x) \rightsquigarrow_{T'} f(y)$ since the rest follows. We have:

$$\begin{aligned} x \rightsquigarrow_T y &\Rightarrow \exists p = (x, \ldots, y) = \{(x, p_2), (p_2, p_3), \ldots, (p_{l-1}, y)\} \subseteq E \\ &\Rightarrow p' = \{(f(x), f(p_2)), (f(p_2), f(p_3)), \ldots, (f(p_{l-1}), f(y))\} \subseteq E' \\ &\Rightarrow f(x) \rightsquigarrow_{T'} f(y) \end{aligned}$$

□

This lemma implies that for most nodes, this move has no effect: If neither x nor y are v or w, this lemma says that their are connected in T' iff they are also connected in T', which one can see in figure 3.2. r and a stay connected after the move, but b and a still are not connected. v and w however swap their connections: For example, w and a are connected in T' since $f(w) = v$ and $f(a) = a$ are connected in T, but v and a are not connected in T' since $f(v) = w$ and a are not connected in T. Now, one can evaluate every possible case for x and y, use the lemma to find the correct value and check that AM::SwapNodes assigns this value.

3.1.3 "Prune and reattach" move

Next, we discuss the "prune and reattach" move, where an entire subtree is moved from one node to another:

Definition 3.6 ("Prune and reattach" move, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree, $v \in V \setminus \{r\}$ and $t \in V$ with $v \not\rightsquigarrow_T t$. We define the mutation tree T' after the "prune and reattach" move as $T' = (V, E', r)$ with

$$E' := (E \setminus \{(p_T(v), v)\}) \cup \{(t, v)\}$$

```
1: function AM::PruneReattach(V, A ∈ {0,1}^{|V|×|V|}, v, t)
2:     A' ← 0 ∈ {0,1}^{|V|×|V|}
3:     for all x ∈ V do
4:         for all y ∈ V do                                  ▷ Unroll completely
5:             if A[v][y] then
6:                 A'[x][y] ← A[x][t] ∨ (A[v][x] ∧ A[x][y])   ▷ See lemma 3.8
7:             else
8:                 A'[x][y] ← A[x][y]                        ▷ See lemma 3.7
9:             end if
10:        end for
11:    end for
12:    return A'
13: end function
```

Algorithm 3.4: Algorithm to perform the "prune and reattach" move on an ancestor matrix. The node v is attached to the node t, assuming that we have $v \not\rightsquigarrow_T t$.

We present the algorithm AM::PruneReattach 3.4 that computes the resulting ancestor matrix of the "prune and reattach" move. It simply iterates over all pairs of x and y and evaluates whether they are connected using two lemmata, which apply to two different cases. In the following, we will explain these cases:

Lemma 3.7. *Let $x, y \in V$ with $v \not\rightsquigarrow_T y$. We have:*

$$x \rightsquigarrow_{T'} y \Leftrightarrow x \rightsquigarrow_T y$$

The idea behind this lemma is that paths in the tree are unharmed if they do not go via the removed edge $(p_T(v), v)$. This is for example the case for r and 2: Since their path in T does not contain the edge (r, v), it is not cut by the move. However, the nodes 2 and t are still not connected in T'.

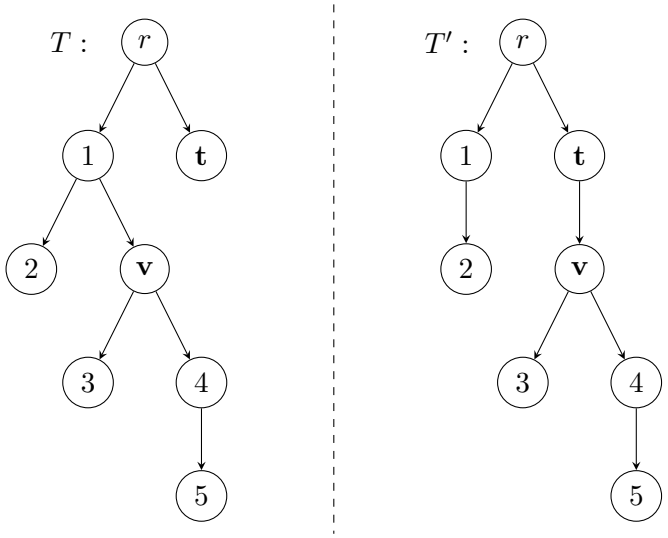


Figure 3.3: Example tree to explain the "prune and reattach" move.

Lemma 3.8. *Let $x, y \in V$ with $v \rightsquigarrow_T y$. We have:*

$$x \rightsquigarrow_{T'} y \Leftrightarrow (v \rightsquigarrow_T x \rightsquigarrow_T y \vee x \rightsquigarrow_T t)$$

The idea of this lemma is that paths over the node v exist in the new tree iff they are either entirely in the subtree below v and are therefore carried over to T', or are newly established via the edge (t, v). Positive examples for the case $v \rightsquigarrow_T x \rightsquigarrow_T y$ are 4 and 5, and for the case $x \rightsquigarrow_T y$ we have r and 3. The direction $\Leftarrow$ is therefore rather simple, but $\Rightarrow$ is not quite obvious. We therefore provide the following formal proof for $\Rightarrow$: Let $p = (p_1, \ldots, p_l) \subseteq E'$ be the path from x to y in T' ($p_1 = x$ and $p_l = y$). We now consider the cases $(t, v) \in p$ and $(t, v) \notin p$: If we have $(t, v) \in p$, we then also have $x \rightsquigarrow_{T'} t$, and with $v \not\rightsquigarrow_T t$ and lemma 3.7 also $x \rightsquigarrow_T t$, one of the options for the right-hand side. If we have $(t, v) \notin p$, we also have $x \not\rightsquigarrow_{T'} v$. From this, we need to conclude $v \rightsquigarrow_{T'} x \rightsquigarrow_{T'} y$ in order to avoid a contradiction with $v \rightsquigarrow_{T'} y$. Lastly, we have $v \rightsquigarrow_T x \rightsquigarrow_T y$ since we neither have $(v, t) \in p$ nor $(p_T(v), v) \in p$.

3.1.4 "Swap unrelated subtrees" move

The next move is a modification of the "prune and reattach" move: Instead of moving a single tree, we now move two trees in the "swap unrelated subtrees" move:

Definition 3.9 ("Swap unrelated subtrees" move, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree, and $v, w \in V \setminus \{r\}$ with $v \neq w$, $v \not\rightsquigarrow_T w$, and $w \not\rightsquigarrow_T v$. We define the mutation tree T' after the "swap unrelated subtrees" as $T' = (V, E', r)$ with

$$E' := (E \setminus \{(p_T(v), v), (p_T(w), w)\}) \cup \{(p_T(w), v), (p_T(v), w)\}$$

Since we have $v \not\rightsquigarrow_T w \Rightarrow v \not\rightsquigarrow_T p_T(w)$ and $w \not\rightsquigarrow_T v \Rightarrow w \not\rightsquigarrow_T p_T(v)$, we could implement the "swap unrelated subtrees" move as two calls to AM::PruneReattach. However, using a custom algorithm makes the implementation of all moves together easier. Therefore, we present the custom algorithm AM::SwapUnrelatedSubtrees 3.5 that computes the resulting ancestor matrix of the entire "swap unrelated subtrees" move. One can simply verify this algorithm by evaluating all cases and checking that two calls to AM::PruneReattach and AM::SwapUnrelatedSubtrees assign the same values. However, we remark that the case

```
1: function AM::SwapUnrelatedSubtrees(V, A ∈ {0,1}^{|V|×|V|}, v, w)
2:     A' ← 0 ∈ {0,1}^{|V|×|V|}
3:     for all x ∈ V do
4:         for all y ∈ V do                                  ▷ Unroll completely
5:             if A[v][y] ∧ ¬A[w][y] then
6:                 A'[x][y] ← A[x][p_T(w)] ∨ (A[v][x] ∧ A[x][y])
7:             else if ¬A[v][y] ∧ A[w][y] then
8:                 A'[x][y] ← A[x][p_T(v)] ∨ (A[v][x] ∧ A[x][y])
9:             else
10:                A'[x][y] ← A[x][y]
11:            end if
12:        end for
13:    end for
14:    return A'
15: end function
```

Algorithm 3.5: Algorithm to perform the "swab subtrees" move for unrelated subtrees on an ancestor matrix. The node v is attached to $p_T(w)$ and the node w is attached to $p_T(v)$, assuming that we have $v \neq w$, $v \not\rightsquigarrow_T w$, and $w \not\rightsquigarrow_T v$.

$v \rightsquigarrow_T y$ and $w \rightsquigarrow_T y$ is impossible since it would lead to either $v \rightsquigarrow_T w \rightsquigarrow_T y$ or $w \rightsquigarrow_T v \rightsquigarrow_T y$, which would contradict our assumption of $v \not\rightsquigarrow_T w$ and $w \not\rightsquigarrow_T v$. It is therefore safe for AM::SwapUnrelatedSubtrees to not catch this case.

3.1.5 "Swap related subtrees" move

The "swap related subtrees" move is the last and one of the most interesting ones:

Definition 3.10 ("Swap related subtrees" move, [JKB16])**.** Let $T = (V, E, r)$ be a mutation tree, $v, w \in V \setminus \{r\}$ with $v \neq w$ and $w \rightsquigarrow_T v$, and $t \in V$ with $v \rightsquigarrow_T t$. We define the mutation tree T' after the "swap related subtrees" move as $T = (V, E', r)$ with

$$E' := (E \setminus \{(p_T(v), v), (p_T(w), w)\}) \cup \{(p_T(w), v), (t, w)\}$$

As for the "swap unrelated subtrees" move, two entire subtrees are moved within the tree. However, we now allow v to be a descendant of w. Therefore, we can not actually swap the subtrees since it would introduce the cycle $w \rightsquigarrow_{T'} p_T(v) \rightsquigarrow_{T'} w$ to the tree. Instead, a third node t is sampled from the descendants of v and w is attached to t. We introduce the algorithm AM::SwapRelatedSubtres (Algorithm 3.6) to compute the resulting ancestor matrix of the move. The algorithm partitions the nodes of the tree into three classes:

$$\begin{aligned} A &= \{x \in V : v \not\rightsquigarrow_T x \wedge w \not\rightsquigarrow_T y\} \\ B &= \{x \in V : v \not\rightsquigarrow_T x \wedge w \rightsquigarrow_T x\} \\ C &= \{x \in V : v \rightsquigarrow_T x\} \end{aligned}$$

Figure 3.4 illustrates the node classes before and after the move. We can therefore argue the algorithm's correctness by providing lemmata for all class combinations. One can then check the correctness by evaluating every combination, using the relevant lemma to find the correct value, and then checking that SC::SwapRelatedSubtrees assigns this value:

Lemma 3.11. *Let $\{x, y\} \subseteq A$, $x, \{x, y\} \subseteq B$, or $\{x, y\} \subseteq C$. We then have $x \rightsquigarrow_{T'} y \Leftrightarrow x \rightsquigarrow_T y$.*

```
1: function Classify(A, v, w, x)
2:     if A[v][x] then
3:         return "C"
4:     else if A[w][x] then
5:         return "B"
6:     else
7:         return "A"
8:     end if
9: end function
10:
11: function AM::SwapRelatedSubtrees(V, A ∈ {0,1}^{|V|×|V|}, v, w, t)
12:     A' ← 0 ∈ {0,1}^{|V|×|V|}
13:     for all x ∈ V do
14:         c_x ← Classify(A, v, w, x)
15:         for all y ∈ V do                                  ▷ Unroll completely
16:             c_y ← Classify(A, v, w, y)
17:             if c_x = c_y ∨ c_x = "A" then
18:                 A'[x][y] ← A[x][y]
19:             else if c_x = "C" ∧ c_y = "B" then
20:                 A'[x][y] ← A[x][t]
21:             else
22:                 A'[x][y] ← 0
23:             end if
24:         end for
25:     end for
26:     return A'
27: end function
```

Algorithm 3.6: Algorithm to perform the "swab subtrees" move for related subtrees on an ancestor matrix. The node v is attached to $p_T(w)$ and the node w is attached to t, assuming that we have $v \neq w$ and $w \rightsquigarrow_T v$.

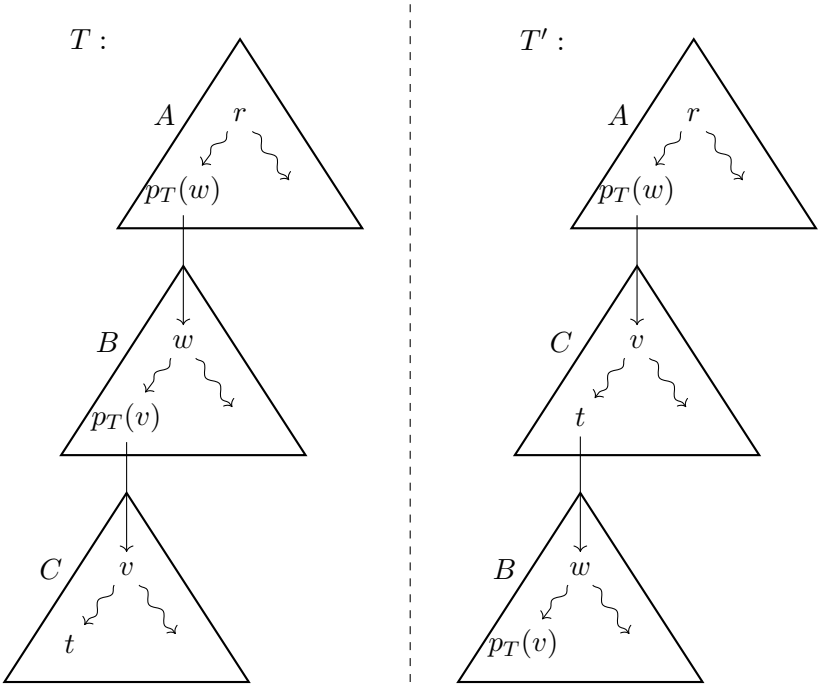


Figure 3.4: Illustration of the tree and the mutation classes before and after the "swap related subtrees" move. Squiggly lines indicate connectivity, while straight lines indicate proper edges.

Proof. This is fairly easy to see in figure 3.4 since an existing path between x and y is entirely inside one of the classes and therefore unaffected by the move, and if there is no path between x and y before the move, then there is no possibility how one of the new edges can establish a new path. □

Lemma 3.12. *Let $x \in A$, and $y \in B \cup C$. We then have $x \rightsquigarrow_{T'} y \Leftrightarrow x \rightsquigarrow_T y$.*

Proof. If there is a path between x and y in T, then there is also a path in T from x in class A to the root of y's class, either B or C. This property is preserved by the move since we can either remove the part of the path through B or add the part through C to construct a path to from x to the relevant root. From there, we can simply add the path from the subtree's root to y to construct the complete path from x to y in T'. We similarly construct a path from x to y in T if there is a path between them in T' and therefore, the equivalence holds true. □

Lemma 3.13. *Let $x \in B$ and $y \in A \cup C$. We then have $x \not\rightsquigarrow_{T'} y$.*

Proof. This is obvious to see in figure 3.4: B is the lowest class in T' and no path leads out if it, so it is impossible to construct a path to a node in A or C. □

Lemma 3.14. *Let $x \in C$ and $y \in A$. We then have $x \not\rightsquigarrow_{T'} y$.*

Proof. The entire class C is below class A, both in T and T'. If there were a path from x to y in either of the trees, there would be a cycle in the tree and therefore, such a path can not exist. □

Lemma 3.15. *If $x \in C$ and $y \in B$, we have $x \rightsquigarrow_{T'} y \Leftrightarrow x \rightsquigarrow_T t$.*

Proof. If there is a path from x to t in T, there is also a path from x to w in T' since t is the new parent of w in T'. Since w is the root of B, there is also a path from w to y and we therefore have $x \rightsquigarrow_{T'} y$. If there is however no path from x to t in T, then it is impossible to reach any node outside of C from x in T'. □

3.1.6 Implementation

All algorithms introduced in the previous section rely on bit-level operations. Most prominently, the algorithms AM::PruneReattach 3.4, AM::SwapUnrelatedSubtrees 3.5 and AM::SwapRelatedSubtrees 3.6 iterate over and set every individual bit in the ancestor matrix. This would be hard to implement efficiently with general-purpose CPUs, but we were able to use the special properties of FPGAs to implement these algorithms efficiently.

First of all, we constrained the maximal number of nodes to a power of two called $n_{\max}$. This number is arbitrary and is fixed to the executable once it is built, but it can be changed in the code by changing one constant. In the final build, we have used $n_{\max} = 64$ since it is the highest value that still resulted in a synthesizable design. Every ancestor matrix therefore technically contains $n_{\max} \times n_{\max}$ entries regardless of the input size and every loop over the set of nodes is executed exactly $n_{\max}$ times. Irrelevant entries are ignored and irrelevant loop iterations perform arbitrary but unharmful operations. This eliminates the possibility to shortcut loops and to improve the performance for small inputs, but it assures that the loops inside the tree scoring kernel do not get out of order. Without fixed trip counts, we could not pipeline the complete design, which is one of the core sources of performance on FPGAs. Continuing with our optimizations, we have encoded ancestor matrices as arrays of $n_{\max}$ $n_{\max}$-bit words, where the ith word describes the descendants of the ith node. Individual words can therefore easily be stored in registers and whole matrices can be stored in RAM blocks. Therefore, the first index of an ancestor matrix access technically addresses a word in a memory block and the second index isolates an individual bit. Next, the memory access patterns of the update algorithms are very easy to predict: Given the previous ancestor matrix A we only need to preload the words $A[v]$ and $A[w]$ and then load the word $A[x]$ for every iteration of the outer for-loop; All queries required by the algorithm can be executed with these three words. The IsParent algorithm 3.2 even needs only two memory accesses in total to operate, namely $A[v]$ and $A[x]$. Lastly, all individual bit operations are independent of each other and so simple that the inner loops of 3.2, 3.4, 3.5 and 3.6 can be fully unrolled. In the end, every update and every parent search can be described as a single loop that loads a word, executes custom, self-contained logic on it, and writes the resulting word back, if necessary. This is perfect for FPGAs since this custom logic can be implemented parallelized on the bit level, while a CPU would need to isolate and shift bits to accomplish this. This loop is also easily pipelinable and requires exactly $n_{\max}$ cycles to complete if its latency can be neglected. As an example, we have provided algorithm listing 3.7 which is an optimized version of AM::PruneReattach 3.4. Lastly, we also needed to merge the update algorithms 3.3, 3.4, 3.5, and 3.6 into one for-loop so that there is only one loop that is always executed instead of four individual loops of which only one is executed, which would inhibit the pipelining of the entire tree scoring kernel.

3.2 Likelihood computation

The likelihood score of a mutation tree is used on multiple occasions, mainly to judge whether a newly proposed tree is the best and to compute the probability of accepting a proposed tree. Since every current tree has been a proposed tree at some point, it is possible to carry the likelihood score along with the tree and only compute the likelihood score of the proposed tree in the chain step. Both our and the reference implementation do this in a fundamentally similar way described in algorithm 3.8: Both implementations iterate over every cell and tree node and identify the mutations the cell would have if it were attached to this node. These mutations yield a likelihood score for the cell-node pair. As shown in lemma 2.9, we have to choose the attachment node for every cell that yields the highest likelihood, which is then called

```
1: function AM::PruneReattach(V, A ∈ {0,1}^{|V|×|V|}, v, t)
2:     A' ← 0 ∈ {0,1}^{|V|×|V|}
3:     p_v ← A[v]
4:     for all x ∈ V do
5:         p_old ← A[x]
6:         p_new ← 0 ∈ {0,1}^{|V|}
7:         for all y ∈ V do                                  ▷ Unroll completely
8:             if p_v[y] then
9:                 p_new[y] ← p_old[t] ∨ (p_v[x] ∧ p_old[y])
10:            else
11:                p_new[y] ← p_old[y]
12:            end if
13:        end for
14:        A'[x] ← p_new
15:    end for
16:    return A'
17: end function
```

Algorithm 3.7: A version of the "prune and reattach" algorithm 3.4 using the optimizations described in section 3.1.6. Reused data is preloaded as early as possible and direct memory accesses are moved out of the inner loop. This loop is then unrolled and forms a custom logic block.

the likelihood of the cell. These likelihoods are then multiplied together to yield the likelihood of the entire tree.

Both implementations try to minimize the number of floating-point operations to save time and preserve precision. Therefore, they do not directly multiply individual likelihood factors together. Instead, they count the occurrences of true and false positives and negatives, raise the probabilities of these events to the counted numbers, and multiply these powers together. SCITE even differentiates between a fast and an accurate computation method: In the fast method, the likelihoods of the cells are directly multiplied together after they are decided, and in the accurate method only the event counts are summed up and the final likelihood score is just computed in the end. SCITE first runs the fast method on the newly proposed tree and if the result is close to or higher than the best-known likelihood score, the accurate method is used again. We chose to only implement and use the fast method since our performance would not gain from using both methods: We would need to allocate chip space to both methods and would probably need to run both every time to avoid diverging loops. Our quality benchmark discussed in section 4.4 assures us that the accuracy of our implementation is still satisfying.

Running many power operations however costs a lot of time, and in the case of FPGAs also a lot of space. Therefore, both implementations work with log-likelihoods instead since power and multiply operations translate to the lightweight multiply and add operations for log-likelihoods. Additionally, log-likelihoods are required since real-world likelihoods often have a very small absolute value that can not be represented even with 64-bit floating point numbers. Using the logarithm of the likelihoods instead brings the absolute values into ranges where even 32-bit floating point numbers are precise enough. Lastly, it is also possible to work with log-likelihoods for most of the algorithm since the logarithm preserves order. Exponentiation of the log-likelihoods is therefore only required when computing the acceptance probability.

There are however two big differences between our implementation and the reference: The reference implementation identifies the mutations of a cell-node pair by walking up the tree from

the attachment node to the root using a parent vector (definition 3.1), just like in the construction function for ancestor matrices (algorithm 3.1). This is not feasible on FPGAs, as we have already discussed in section 3.1. Instead, our implementation iterates over all genes and uses an ancestor matrix to check whether this gene is an ancestor of the attachment node and therefore mutated. The tradeoffs of iterating over all nodes/genes instead of walking up a tree are already discussed in subsection 3.1.6; In summary, it improves the structural performance of the design but it does not scale up for smaller inputs. The next difference is in how the resulting values are reduced: The reference implementation directly reduces found values by always keeping the best-known likelihood of a cell-node pair and an accumulator for the final likelihood. We encountered feedback problems with this approach which inhibited loop pipelining. Therefore, we chose a clear map-reduce approach: We first compute and store the likelihood of every cell-node pair in a buffer, max-reduce the node axis and then sum-reduce the cell axis. Every one of these three operations is executed by separate, independent loops, which are perfectly pipelinable and unrolled by at least one dimension. Additionally, we were able to coalesce and partially unroll the two remaining dimensions of the likelihood mapping loop. However, this obviously increases the memory and chip space requirements of our design. Our resulting log-likelihood computation algorithm is listed as algorithm 3.9.

function Likelihood(C, G, $T = (V, E, r)$, $d \in \{0, 1, 2\}^{|C| \times |G|}$, $\{\alpha, \beta\} \subseteq [0, 1]$)
  $l_T \leftarrow 1$
  for all $c \in C$ **do**
    $l_c \leftarrow 0$
    for all $v \in V$ **do**
      $l_{c,v} \leftarrow 1$
      for all $g \in G$ **do**
        $l_{c,v} \leftarrow l_{c,v} \cdot \lambda(d_{c,g}, g \rightsquigarrow_T v)$
      end for
      if $l_{c,v} \geq l_c$ **then**
        $l_c \leftarrow l_{c,v}$
      end if
    end for
    $l_T \leftarrow l_T \cdot l_c$
  end for
  return l_T
end function

Algorithm 3.8: Formal algorithm to compute $\Lambda_d(T)$ (Definition 2.2, Lemma 2.9) for a given mutation tree T, a mutation data matrix d and error probabilities α and β.

3.3 Move Proposals

Another area of concern in the design of ffSCITE was the sampling of random experiments to propose state changes. On multiple occasions in the ChainStep algorithm (Algorithm 2.1), parameters of later actions are sampled uniformly from a set of options and a Bernoulli trial decides whether the newly proposed state is accepted as the new current state. These samples require a source of randomness, and Jahn et al. [JKB16] used the `rand` function from the standard C library for their reference implementation. This function is however not available on FPGAs, so we had to use other methods. The C++ standard library provides a framework for handling random numbers distinguishes uniform random number generators (URNGs), which

function LogLikelihood(V, C, G, $a \in \{0,1\}^{|V|\times|V|}$, $d \in \{0,1,2\}^{|C|\times|G|}$, $\{\alpha,\beta\} \subseteq [0,1]$)
  $l_{\text{pair}} \leftarrow 0 \in (-\infty, 0]^{|C|\times|V|}$
  for all $(c,v) \in C \times V$ **do** ▷ Unroll as much as possible
    occurrences $\leftarrow 0 \in \mathbb{N}_0^{3\times 2}$
    for all $g \in G$ **do** ▷ Unroll completely
      posterior $\leftarrow d[c][g]$
      prior $\leftarrow a[g][v]$
      occurrences[posterior][prior] $\leftarrow$ occurrences[posterior][prior] $+ 1$
    end for
    $l_{\text{pair}}[c][v] \leftarrow \log(1-\alpha) \cdot$ occurrences[0][0]
    $l_{\text{pair}}[c][v] \leftarrow l_{\text{pair}}[c][v] + \log(\alpha) \cdot$ occurrences[1][0]
    $l_{\text{pair}}[c][v] \leftarrow l_{\text{pair}}[c][v] + \log(\beta) \cdot$ occurrences[0][1]
    $l_{\text{pair}}[c][v] \leftarrow l_{\text{pair}}[c][v] + \log(1-\beta) \cdot$ occurrences[1][1]
  end for
  $l_{\text{cell}} \leftarrow 0 \in (-\infty, 0]^{|C|}$
  for all $c \in C$ **do**
    $l_{\text{cell}}[c] \leftarrow \max\{l_{\text{pair}}[c][v] : v \in V\}$ ▷ Unrolled for-loop in maximum operation
  end for
  return $\sum_{c\in C} l_{\text{cell}}$ ▷ Unrolled for-loop in sum
end function

Algorithm 3.9: Our FPGA-optimized version of algorithm 3.8 to compute $\log(\Lambda_d(T))$ for a given mutation tree T encoded as an ancestor matrix a, a mutation data matrix d and error probabilities α and β.

are pseudo-random algorithms that sample uniformly from a set of integers fixed by their design, and distributions, which are algorithms that take the output of any URNG to provide a sample of the requested distribution. This may be a uniform distribution from a smaller or bigger set than the original output, but it may also be a Bernoulli trial or a normal distribution. The C++ standard library often provides only a single class for every distribution and since those that are relevant to us work just fine on FPGAs, we did not have to deal with this issue.

Deciding on a URNG however was a big decision: There are different generators in the C++ standard library. These URNGs have different statistical properties, but some are more or less practical as a chip design, at least out of the box. As it is common practice, we decided to initially use a simple URNG that may or may not be perfect for our use case and to exchange it once we encountered problems either with its output quality or performance. This URNG is `oneapi::dpl::minstd_rand`, an implementation of the Minstd URNG [PMS93] provided by Intel in the OneDPL library; A subset of the C++ standard library designed especially for and with OneAPI. Minstd is an improved version of the peer-reviewed Minstd0 URNG [PM88], which was published by the same authors as a technical correspondence. Park and Miller, the authors of Minstd0, advertised it with the words "We believe that this is the generator that should always be used- unless one has access to a random number generator known to be better." [PM88] Since the implementation also does not have any internal loops which could cause structural problems, this URNG was a good first choice for us. Other, more popular URNGs like the Mersenne Twister [MN98] did not meet this requirement and were therefore discarded for our initial implementation. Finding a better URNGs, however, proved to be harder than expected. We found it hard as an outsider to identify the state of the art in the field and since ffSCITE already passes our quality benchmarks with Minstd, we decided to prioritize other, more pressing issues discussed in the previous sections and keep working with Minstd.

There are however problems with Minstd that we had to work around: Minstd is a Lehmer random number generator [Leh51]. This means that given a previous output or seed X_k, its next output is defined as $X_{k+1} = a \cdot X_k \mod m$ for well-chosen parameters a and m. In the case of Minstd, these are $a = 48271$ and $m = 2^{32} - 1 = 2147483647$. An implementation of Minstd, therefore, needs to work with integers wider than 32 bits, and since `oneapi::dpl::minstd_rand` is designed to work on CPUs, GPUs and FPGAs, it uses 64-bit integers. 64-bit modulo operations are however *very* resource-intensive on FPGAs and require multiple cycles to complete. Lastly, the output of Minstd is supposed to appear random and is fed back as the new state of the generator. It is therefore impossible to predict the new state of the URNG and any loop that draws from a URNG has a data dependency that increases its II significantly. We were not able to solve this problem as it would require researching or developing another URNG and, therefore, needed work around the issue by isolating the move proposals to a custom kernel, reducing the total calls to the URNG, and using multiple independent URNG instances to reduce the critical path.

3.4 Input, Output and Feedback

The change proposer kernel and the internal loops of the tree scorer kernel form a macro-pipeline that processes states in independent steps: For example, the change proposer may already sample a move for a new state while the tree update loop computes the new ancestor matrix of the previous state. This macro-pipeline has to be filled at all times to utilize the entire design and achieve maximum performance. Ensuring this is the task of the IO kernel.

From an application's point of view, the design is supposed to iterate n_{steps} times on the same state and then generate a new, completely random initial state to work with, up to n_{chains} times. Early on, we decided to not generate the initial states on the device since this is a rare task compared to the chain steps. We, therefore, assumed that allocating precious resources to this task would not accelerate the application. Instead, our application's host generates n_{chains} initial states and stores them in global buffers that are transferred to the FPGA's memory. A naïve approach would load a state from the global buffer, send it to the change proposer kernel and wait until the tree scorer is finished to write the result back. The disadvantages of this approach are obvious and lead to low hardware utilization and high latency due to the external memory accesses. However, we were able to exploit three properties of the problem and the hardware design to optimize the occupancy: First, the intermediate and final states of the chains are irrelevant to the algorithm. Only the current state is needed to compute the next state and only the best state needs to be saved and exported. Once a current state has been replaced, it is not needed anymore. Second, nothing in the macro-pipeline needs to know to which step of which chain the processed state belongs. It simply executes the same operations on the initial state of the first chain as it does on the 42,000th step of the 17th chain. Third, the macro-pipeline has a capacity. This means that the IO kernel can send a certain number of states to the change proposer before it needs to start to receive states from the tree scorer. From the IO kernel's point of view, these states are "stored" in the pipeline.

Our final IO kernel (algorithm 3.10) works as follows: First, it assumes that the pipeline has a capacity of c states. Therefore, it initially sends c states from the initial states buffer to the change proposer and waits for the tree scorer to finish the first step. Then, the resulting state from the tree scorer is fed back to the change proposer as often as requested by the user. Once the requested steps are completed, the final chain state is discarded and a new initial state is sent to the change proposer instead. Once all steps of all chains are completed, the kernel flushes the pipeline and halts. This design has multiple advantages over the naïve approach: Firstly, this design is simple. Its logic is simple and its actions are easy to predict, and there are

no loop-carried dependencies. Its biggest advantage however is that it only reads from global memory and does not write back. This simplifies linking as well as routing and placing, which the current FPGA compilers still have problems with sometimes and takes a lot of time. This has been entirely avoided here. Lastly, it might also reduce the energy consumption of the design since data only moves within the chip, but we have not analyzed this. However, this design has the disadvantage that one needs to compile the design multiple times to find the right capacity c, which takes additional effort and CPU-hours for compilation.

for all $i \leq n_{\text{chains}} \cdot n_{\text{steps}} + c$ **do**
  if $i \geq c$ **then**
    $s_{\text{out}} \leftarrow$ Read state from tree scorer kernel
  end if
  if $i < n_{\text{chains}} \cdot n_{\text{steps}}$ **then**
    if $i \mod (n_{\text{steps}} \cdot c) < c$ **then**
      $s_{\text{in}} \leftarrow$ Read next initial state from global memory
    else
      $s_{\text{in}} \leftarrow s_{\text{out}}$
    end if
    Send s_{in} to change proposer kernel
  end if
end for

Algorithm 3.10: Behavioral code of the IO kernel, assuming that the pipeline has a capacity of c states and that the user has requested to simulate n_{chains} chains with n_{steps} steps each.

4 Evaluation

The goal of this thesis can be summarized as implementing the SCITE algorithm with the same solution quality as the reference implementation, but faster. We describe our design choices in the previous chapter 3, but we also needed to verify that these choices lead to this goal. This chapter describes how we tested the solution quality of ffSCITE using the TOST procedure [Sch87] and evaluated its performance using the synthesis report, benchmarking, and profiling.

Before we can discuss the features of the final build of ffSCITE, we need to address a general issue: In section 3.1, we have decided to limit the maximal number of cells and genes that are processable by the compiled design to an arbitrary number. Therefore, we need to decide on a number of cells and genes for our final design. Our final build of ffSCITE supports 64 cells and 63 genes, which is equivalent to mutation trees with 64 nodes. This covers all but one of the example datasets that is bundled with the source code of SCITE and compiling the design for 128 cells and 127 genes leads to internal compiler errors we were not able to resolve in time. Lastly, targeting 64 cells and 63 genes also leads to reasonable resource usage which we will discuss later. Therefore, we decided to continue with this input size.

In this chapter, we will first discuss the properties that can be evaluated using the final synthesis report, namely the hardware usage and loop characteristics. We are then using this information to predict the performance of the design and benchmark both SCITE and ffSCITE to verify this prediction and to evaluate the throughput difference between them. Lastly, we evaluate the solution quality of ffSCITE, for which we have to introduce and execute a statistical test. The most important results of these sections are listed in table 4.1 for quick look-up.

4.1 Hardware usage

Tables 4.2 and 4.3 list the hardware usages of the different kernels for our target FPGA and input sizes. The overall design is very logic-intensive, which makes sense since most of the work is done with custom logic and very few floating-point operations. Of all individual components indicated in the design schematic (Figure 3.1), the tree score mapping loop is the heaviest and makes up roughly 37.7% and 37.5% of the entire design's LUT and FF usage, respectively. This makes sense since it is a three-dimensional loop where the inner-most loop is completely unrolled (i.e. unroll factor 64) and the middle loop is eight times partially unrolled, as we have discussed in section 3.2. Therefore, the inner loop's body is replicated 512 times, which leads to the reported resource usage. The next-biggest component is the change proposer with 33.3% and 25% of the design's LUT and FF usage, respectively. This resource usage is independent of the input size

Metric	Value
Max. no. of cells	64
Max. no. of genes	63
Clock frequency	295.83 MHz
Macro-pipeline capacity	6 states
Device utilization (Kernels + DSP)	69% LUTs
Predicted throughput	577.79 ksteps/s
Measured throughput	566.72 ksteps/s
Maximum speedup	8.63
FPGA	Intel Stratix 10 GX 2800
Board	Bittware 520N
oneAPI/SYCL version	2022.2.0 Build 133.4
Quartus version	20.4.0 Build 72 Pro
BSP version	20.4.0_hpc

Table 4.1: Quick facts about the final ffSCITE build.

	LUTs	FFs	RAMs	MLABs	DSPs
Pipe resources	90	1.9k	0	0	0
IO-Kernel	12.3k	3.9k	180	431	1.5
Tree Scorer	537.4k	817.7k	2.0k	1.6k	340.5
- Find Parent Loop	34.1k	48.2k	22	46	0
- Tree Update Loop	54.1k	61.5k	131	32	0
- Tree Score Mapping Loop	313.4k	439.9k	332	262	72
- Tree Score Reduction Loop	31.5k	70.3k	309	32	0
Change Proposer	274.1k	307.4k	114	299	40.5
Kernels	**843.8k**	**1.2M**	**2.4k**	**2.4k**	**382.5**
Static Partition	**455.2k**	**910.5k**	**2627**	**0**	**1.0k**

Table 4.2: Hardware usage of ffSCITE in absolute numbers, supporting up to 64 cells and 63 genes, as reported in the "Area Analysis" report.

since it is dominated by the URNG executions which involve a 64-bit integer multiplication and remainder operation; Compiling the design for smaller input sizes yields similar resource usages for the change proposer.

It may be possible to further unroll the middle loop of the tree score mapping loop and double its potential throughput and hardware usage, and an extension to bigger inputs may be possible if the compiler issues were resolved. However, experience has shown us that designs with a total logic usage over 75% start to run into issues regarding compilation times and clock frequencies. Therefore, we decided to settle with this configuration. It is however impossible to replicate the entire design another time since we lack the resources.

4.2 Throughput prediction

In this section, we discuss the "Throughput Analysis" of the synthesis report to predict the design's throughput; A simplified version of this analysis is provided in table 4.4. We have already established that the design has multiple components that work independently and form a macro-pipeline. If this macro-pipeline is always saturated, we can disregard the latency of the pipeline and the throughput of the design should be equal to the component with the lowest

	LUTs	**FFs**	**RAMs**	**MLABs**	**DSPs**
Pipe resources	<1%	<1%	0%	0%	0%
IO-Kernel	1%	1%	2%	<1%	<1%
Tree Scorer	29%	22%	17%	2%	6%
- Find Parent Loop	2%	1%	<1%	<1%	0%
- Tree Update Loop	3%	2%	1%	<1%	0%
- Tree Score Mapping Loop	17%	12%	3%	0%	1%
- Tree Score Reduction Loop	2%	2%	3%	<1%	0%
Change Proposer	15%	8%	1%	<1%	1%
Kernels	**45%**	**32%**	**20%**	**3%**	**7%**
Static Partition	**24%**	**24%**	**22%**	**0%**	**18%**

Table 4.3: Hardware usage of ffSCITE relative to the resources available on Intel Stratix 10 GX 2800 FPGAs, supporting up to 64 cells and 63 genes, as reported in the "Area Analysis" report.

Kernel/Loop	**II**	**No. of iterations (unrolled)**
IO-Kernel	1	$n_{\text{chains}} \cdot n_{\text{steps}} \cdot c$
- Tree Receive Loop	1	64
- Tree Send Loop	1	64
Change Proposer Kernel	263	$n_{\text{chains}} \cdot n_{\text{steps}}$
- Tree Receive Loop	1	64
- Tree Send Loop	1	64
Tree Scorer Kernel	1	$n_{\text{chains}} \cdot n_{\text{steps}}$
- Tree Receive Loop	1	64
- Find Parent Loop	1	64
- Tree Update Loop	1	64
- Tree Score Mapping Loop	1	512
- Tree Score Reduction Loop	1	64
- Tree Send Loop	1	64

Table 4.4: (Simplified) throughput analysis table with the number of iterations for the unrolled loops.

throughput, or the highest required number of cycles required to process one chain step. We, therefore, walk through all relevant points of the throughput analysis to find this theoretical bottleneck.

All kernels have an outer loop that generally executes one chain step per iteration. The IO kernel's outer loop also has a warm-up and cool-down phase to fill and flush the macro-pipeline, but this is negligibly small compared to the total number of chain steps. The outer loops of the IO kernel and the tree scorer kernel have an II of 1, but the change proposer kernel's outer loop has an II of 263 due to the data dependency discussed in section 3.3. This means that the outer loops of the IO and tree scorer kernels would be able to process one chain step per cycle, but the change proposer is only able to process a chain step every 263 cycles. Therefore, our first bottleneck is the change proposer kernel with 263 cycles per chain step.

Within these outer loops, there is also one input loop and one output loop per kernel that send and receive ancestor matrices word by word. This is necessary since transmitting an ancestor matrix in parallel would require a $64^2 = 4096$-bit wide pipe, which has caused timing issues during compilation. These RX/TX loops execute one iteration per word and therefore require 64 cycles per chain step. This also applies to the Find Parent loop and the Tree Update loop since they iterate over every one of the 64 nodes in a mutation tree, as well as the Tree Score Reduction loop since it is unrolled over one dimension and therefore also requires 64 cycles per chain step. Therefore, the change proposer remains as the current bottleneck. The only remaining loop in our discussion is the tree score mapping loop. It is a three-dimensional loop that iterates over all cells, nodes, and genes. The gene dimension is however completely unrolled and the node dimension is unrolled eight times and coalesced with the cell dimension. The remaining implementation is therefore a one-dimensional loop that requires $\frac{64 \cdot 64}{8} = 512$ cycles for one chain step. This is the new bottleneck and therefore, we predict that the design will finish one chain step every 512 cycles. Together with the final clock frequency of 295.83 MHz, we predict that ffSCITE's chain step throughput should be

$$\frac{295.83 \cdot 10^6 \text{ cycles}}{1 \text{ s}} \cdot \frac{1 \text{ steps}}{512 \text{ cycles}} \approx 577.79 \text{ ksteps/s}$$

Since there are no caches that can warm up and cool down and since the number of iterations does not change with the input sizes, we predict that ffSCITE will always achieve this throughput regardless of the input size and the number of chain steps.

4.3 Throughput benchmark

We verified our performance goal by running SCITE and ffSCITE in different configurations and comparing their makespan. Three dimensions may impact the makespan of the SCITE algorithm: The input size, i.e. the number of cells and genes included in the input, the number of chains to execute, and the number of steps to execute per chain. We used three random inputs, one with 16 cells and 15 genes, one with 32 cells and 31 genes, and one with 64 cells and 63 genes. Our options for the number of chains were 6 and 12, and we used all chain lengths from 500,000 steps to 2,000,000 steps in increments of 500,000 steps. We also let every application run ten times for every parameter combination and averaged out the resulting makespan. We should also note that ffSCITE measures the makespan of the FPGA kernels and therefore excludes the time to read inputs, generate initial states and emit the resulting tree. SCITE however measures the runtime of the entire application, which introduces some noise to the measurements, especially for small parameters. However, these effects should be negligible for big parameters.

We have compiled both SCITE and ffSCITE with the Intel DPC++ compiler version 2022.1.0 provided by Intel oneAPI 2022.1.0, which supports SYCL version 2022.2 and internally uses Intel

Quartus version 20.4.0 build 72 Pro. The precise compiler flags for ffSCITE were

```
-fintelfpga -qactypes -DHARDWARE -Xshardware -Xsv -Xsparallel=8 -Xsprofile
                          -Xshigh-effort -Xsseed=1
```

and those for SCITE were

```
-O3 -DNDEBUG -qactypes -fintelfpga -Wtautological-constant-compare -MD -MT
```

Our target FPGA is the Intel Stratix 10 GX 2800 on a Bittware 520N card, using the board support package (BSP) version 20.4.0_hpc, and our target CPU system is a Noctua 2 compute node, which contains two AMD Milan 7763 CPUs and 128 GB of main memory, although SCITE only uses a single core.

The resulting performance metrics are plotted in figure 4.1. ffSCITE's throughput remains constant for all parameter combinations, as we have predicted. Therefore, ffSCITE's graphs are dashed lines since they would otherwise overlap. The exact throughput of ffSCITE is 566.72 ksteps/s, which is 98.1% of the performance we predicted. An additional profiling run using Intel VTune also reveals that the loop with the highest occupation with 99.8% is the Tree Scorer loop, again as predicted. The next most-occupied loops are the other internal loops of the tree scorer with 14.0%, which roughly matches the anticipated occupation of $\frac{512}{64} = 12.5\%$. We therefore still consider our model a match since we generally see the predicted behavior, within some errors.

The performance metrics of SCITE however have more variation than ffSCITE, which is expected: It only iterates through its loops as often as necessary, which means that the throughput of SCITE in steps per second is bigger for smaller inputs. There is also some noise in the measurements, both due to the previously mentioned measurement scope as well as warming caches and interactions with the operating system. One effect that we however do not and do not try to understand is that for the small input, SCITE's throughput appears to decrease with an increasing number of steps. However, the throughput is nearly constant for the mid-sized and big input and we can therefore reliably deduce that ffSCITE has an up to 8.53 times higher throughput than SCITE. It might also be interesting to further explore the effects of different input sizes and chain counts. However, this would require an in-depth analysis of SCITE's performance, which is outside the scope of this thesis.

We had also set ourselves the optional goal to obtain a higher throughput than Ernst et al. [EGJW20]. We were not able to properly verify this since we neither have access to their source code nor did they list absolute performance figures or the total single-thread speedup compared to the reference implementation. However, we assume that their throughput is higher since their first single-threaded optimization alone achieved a speedup of 6.4 compared to the reference and they were able to use multiple threads to increase their throughput by a factor of 61.3 compared to their single-threaded throughput.

4.4 Quality Testing

We now look at the solution quality. Although we used the unit testing library Catch2 to verify individual components, this does not suffice to verify that the entire application behaves correctly. This however can not be done with deterministic tests since SCITE is a stochastic algorithm: Given different seeds, it produces different solutions which may or may not be perfect. We also could not implement ffSCITE as a bit-exact copy of SCITE, especially since we could not use the same URNG as the reference implementation. We, therefore, needed to introduce a statistical test that verifies that given the same input, both SCITE and ffSCITE produce equivalent outputs. In this section, we discuss how we designed the test and what the results are.

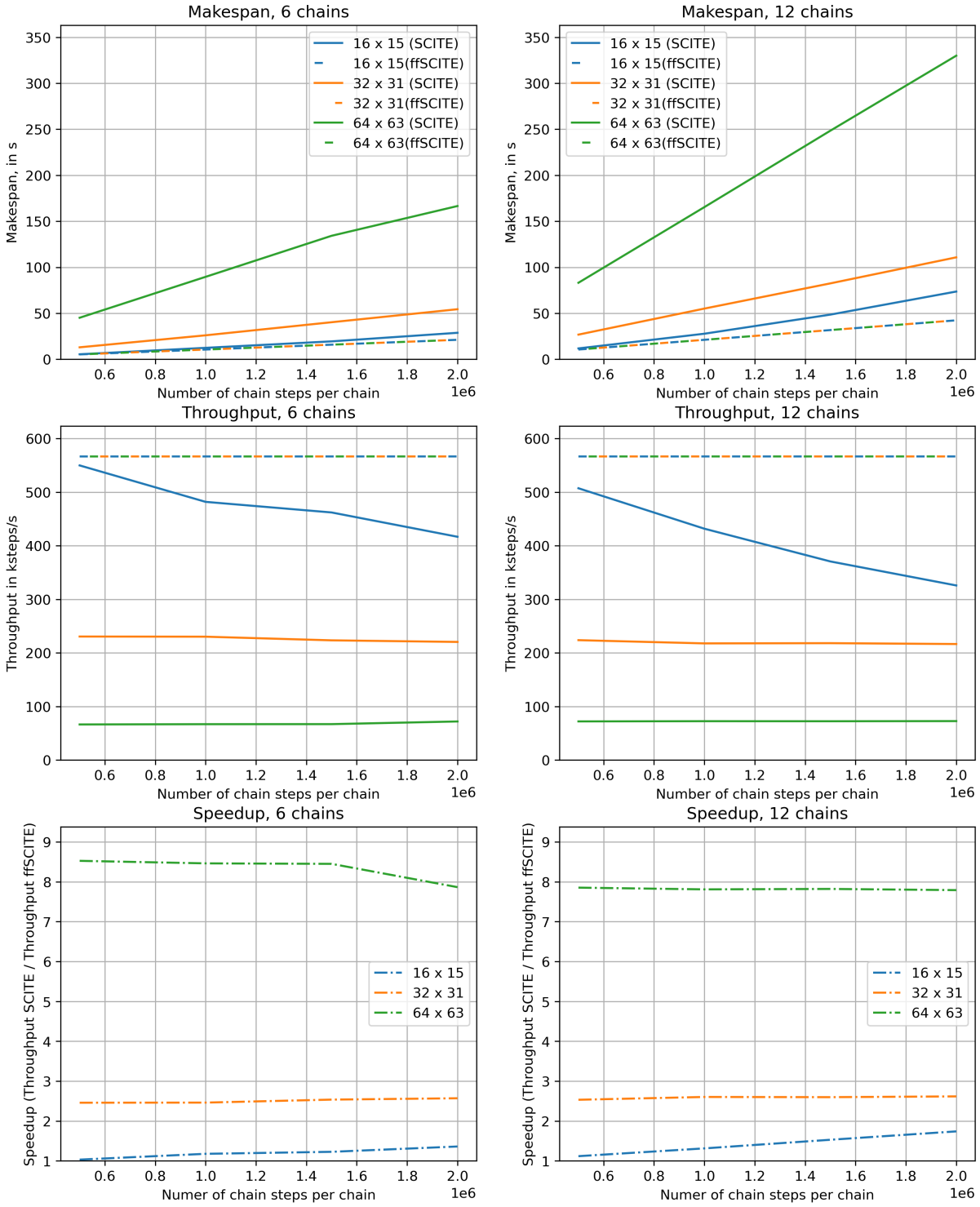


Figure 4.1: Performance metrics of ffSCITE and SCITE. The colors indicate the input size, while the dashed lines indicate ffSCITE's metrics and the solid lines indicate SCITE's metrics. The speedup graphs indicate SCITE's makespan divided by ffSCITE's makespan for the respective inputs.

4.4.1 Methods

Let C be the set of considered cells, G be the set of considered genes, and $d \in \{0,1,2\}^{|C|\times|G|}$ the input mutation data matrix. We model the output mutation trees of SCITE and ffSCITE given the inputs C, G, and d as the random variables T_{SCITE} and T_{ffSCITE}. We again work with log-likelihoods since the true likelihood scores for practically sized inputs are too close to 0 to express them using 64-bit floating-point numbers. Therefore, we intuitively want to find evidence for the statement $\mathbb{E}\log\Lambda_d(T_{\text{SCITE}}) = \mathbb{E}\log\Lambda_d(T_{\text{ffSCITE}})$. However, classical hypothesis tests are only able to provide evidence for inequality statements like $\mathbb{E}\log\Lambda_d(T_{\text{SCITE}}) < \mathbb{E}\log\Lambda_d(T_{\text{ffSCITE}})$. This is not what we aimed for and not what we claim. Therefore, we need a different approach.

In the field of clinical studies, non-inferiority and equivalence trials are a common tool. There, these kinds of trials "are intended to show that the effect of a new treatment is not worse than that of an active control by more than a specified margin." [Sna00] Although there are some complications with using non-inferiority trails for drug testing as pointed out by Snapinn [Sna00], these complications do not apply to our scenario. The exact procedure that we use is called "Two One-Sided t-Tests" (TOST), which was introduced by Schuirmann in 1987 [Sch87] and it requires its designer to set an equivalence margin δ, where absolute differences below this margin are considered negligible. Then, our alternative and null hypotheses are:

$$
\begin{aligned}
H_1 &: \mathbb{E}|\log\Lambda_d(T_{\text{SCITE}}) - \log\Lambda_d(T_{\text{ffSCITE}})| < \delta \\
H_0 &: \mathbb{E}|\log\Lambda_d(T_{\text{SCITE}}) - \log\Lambda_d(T_{\text{ffSCITE}})| \geq \delta \\
&\Leftrightarrow \mathbb{E}\left(\log\Lambda_d(T_{\text{SCITE}}) - \log\Lambda_d(T_{\text{ffSCITE}})\right) \leq -\delta \vee \mathbb{E}\left(\log\Lambda_d(T_{\text{SCITE}}) - \log\Lambda_d(T_{\text{ffSCITE}})\right) \geq \delta
\end{aligned}
$$

Once our hypotheses are established, we can execute a one-sided t-test for each half of H_0 and if both fail, we have provided evidence for the equivalence for ffSCITE and SCITE.

Intuitively, we want to pick δ as close to 0 as possible, but we still want to provide a reasonable margin. A natural choice for δ might be the maximum likelihood difference from one chain step to the next. If ffSCITE were equivalent to SCITE within this margin, it would mean that ffSCITE may have only missed the last chain step to the optimal solution, which one could deem as negligible. However, the effects of a chain step may be chaotic since the cell-node attachments are recomputed and might change after every operation. These attachment changes not only depend on the mutation tree, but also the error probabilities and the observed mutation data. As of this writing, we do not have enough understanding of a chain step's effect on the state likelihood, and we also did not have the time to develop it. Therefore, we decided to work on the mutation matrix level instead. This is also justified by the fact that we have assigned the likelihood score in definition 2.2 to mutation matrices, not to mutation trees. These are only a tool to ease the proposal of mutation matrices. Therefore, we have decided to set our δ to the maximum log-likelihood difference induced by a bit-flip, the smallest logical difference between two bit matrices. According to lemma 4.1, this is

$$\delta := \max\{|\log(\alpha) - \log(1-\beta)|, |\log(\beta) - \log(1-\alpha)|\}$$

where α and β are the probabilities for false positives and negatives, respectively.

Lemma 4.1. *Let C be the set of considered cells, G be the set of considered genes, $d \in \{0,1,2\}^{|C|\times|G|}$, and $e, e' \in \{0,1\}^{|C|\times|G|}$ where e is arbitrary and e' is defined as:*

$$e'_{c,g} := \begin{cases} \overline{e_{c,g}} & c = x \wedge g = y \\ e_{c,g} & \textit{else} \end{cases}$$

for some $x \in C$, $y \in G$. In other words, e' is a version of e where the bit at position (x, y) is flipped. Let also $\alpha, \beta \in [0, 1]$ be the probabilities of false positives and negatives, respectively. Then, we have:

$$|\log \Lambda_d(e) - \log \Lambda_d(e')| \leq \max\{|\log(\alpha) - \log(1-\beta)|, |\log(\beta) - \log(1-\alpha)|\}$$

where Λ_d is the likelihood function of definition 2.2. In other words, the difference in log-likelihood induced by a single bit flip is equal to or less than $\max\{|\log(\alpha) - \log(1-\beta)|, |\log(\beta) - \log(1-\alpha)|\}$.

Proof. First of all, we should remark that we have

$$\begin{aligned} \log \Lambda_d(e) &= \sum_{c \in C} \sum_{g \in G} \log \lambda(d_{c,g}, e_{c,g}) \\ \Rightarrow |\log \Lambda_d(e) - \log \Lambda_d(e')| &= |\log \lambda(d_{x,y}, e_{x,y}) - \log \lambda(d_{x,y}, e'_{x,y})| \\ &= |\log \lambda(d_{x,y}, e_{x,y}) - \log \lambda(d_{x,y}, \overline{e_{x,y}})| \end{aligned}$$

If we have $d_{x,y} = 2$, we therefore have

$$|\log \Lambda_d(e) - \log \Lambda_d(e')| = |\log(1) - \log(1)| = 0$$

Now, one can consider all combinations for $d_{x,y}$ and $e_{x,y}$ to find that we either have

$$|\log \Lambda_d(e) - \log \Lambda_d(e')| = |\log(1-\alpha) - \log(\beta)|$$

or

$$|\log \Lambda_d(e) - \log \Lambda_d(e')| = |\log(1-\beta) - \log(\alpha)|$$

and therefore

$$|\log \Lambda_d(e) - \log \Lambda_d(e')| \leq \max\{|\log(\alpha) - \log(1-\beta)|, |\log(\beta) - \log(1-\alpha)|\}$$

□

Let us summarize our quality testing method: First, we have randomly generated 64 observed mutation data matrices considering 64 cells and 63 genes, with a probability of $\alpha := 10^{-6}$ for false positives, $\beta := 0.25$ for false negatives, and 0.25 for missing data. Then, we ran 6 chains á 1,000,000 steps on these mutation data matrices, using both SCITE and ffSCITE, and stored the resulting max-likelihood mutation trees. This input size of 64 cells and 63 genes is the maximum our build of ffSCITE can support. We chose the number of input sets, chains, and steps as a compromise between expressiveness and execution time since we wanted to run the same experiment multiple times during development to continuously check our design. The error probabilities are however arbitrary since we did not have realistic numbers available or the time to research them. After the execution was completed, we used a Python script to compute the differences in log-likelihood produced by SCITE and ffSCITE for the same inputs and used SciPy to run one-sided t-tests against the parts of our null-hypothesis with a significance level of 0.01. Therefore, the significance level of the entire test is 0.02.

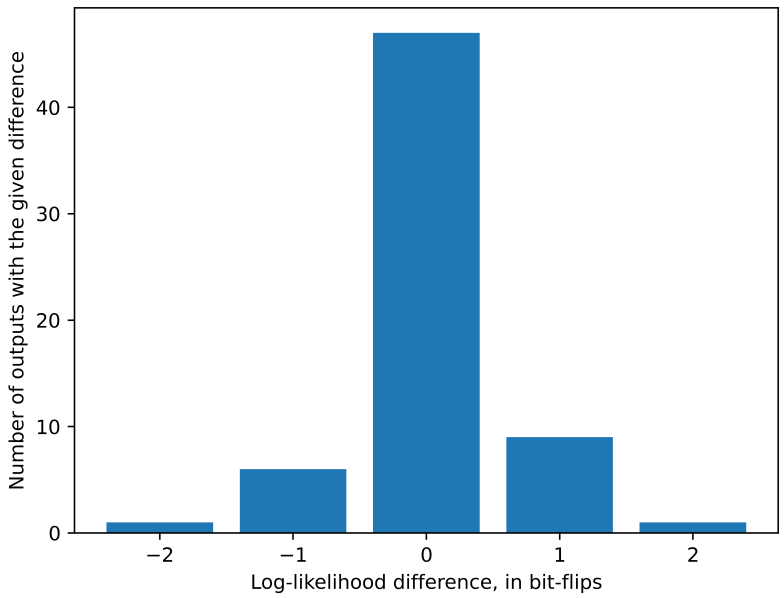


Figure 4.2: Accumulated differences in log-likelihood between solutions provided by SCITE and ffSCITE. A positive difference means that ffSCITE has provided a better solution than SCITE, and a negative difference means the opposite.

4.4.2 Results

Out of the 64 inputs, SCITE and ffSCITE achieved the same likelihoods for 47 inputs. ffSCITE scored better in 10 cases, and SCITE scored better in 7 cases. Interestingly, all encountered differences are multiples of our δ, which is defined as the maximum difference induced by a bit-flip. Therefore, we plotted the number of outputs with a certain log-likelihood difference in multiples of δ or bit-flips in figure 4.2. Another notable finding is that SCITE and ffSCITE never deviate by more than two bit-flips and the mean difference between SCITE's and ffSCITE's log-likelihood score is 0.65. Lastly, the p-value for the test of $\mathbb{E}\left(\log \Lambda_d(T_{\text{SCITE}}) - \log \Lambda_d(T_{\text{ffSCITE}})\right) \leq -\delta$ is $p_- \approx 9.47 \cdot 10^{-21}$ and the p-value for the test of $\mathbb{E}\left(\log \Lambda_d(T_{\text{SCITE}}) - \log \Lambda_d(T_{\text{ffSCITE}})\right) \geq \delta$ is $p_+ \approx 7.87 \cdot 10^{-19}$. Since both of them are lower than our significance level of 0.01, we have to reject H_0 and have provided evidence that ffSCITE and SCITE produce solutions of equivalent quality.

5 Conclusion

ffSCITE reaches the goal of being an implementation of the SCITE algorithm for FPGAs with higher throughput than the reference implementation by Jahn et al. [JKB16]. This algorithm computes the most likely mutation history of a group of tumor cells, given noisy information about their mutation status.

Our biggest contribution is an improvement to the mutation tree encoding: The reference implementation uses parent vectors as the canonical data structure, which contain the parent of every node in the tree. However, it also uses ancestor matrices to analyze the mutation tree, which denote whether there is a path from one node in the tree to another. These ancestor matrices are constructed from parent vectors and this operation is not practical on FPGAs. We, therefore, developed algorithms to reconstruct the mutation tree from an ancestor matrix and to compute the resulting ancestor matrix of a tree modification based on the old one. These algorithms are impractical on CPUs but work well on FPGAs due to their flexibility regarding custom logic, and due to this contribution, our implementation can work exclusively on ancestor matrices and avoid constructing ancestor matrices on the device.

Additionally, we reformulated the algorithm to compute a mutation tree's likelihood: The reference implementation uses two different implementations with different precisions to improve performance, but since ffSCITE would not benefit from computing it twice, we based our formulation on the fast method. The likelihood computation algorithm can be described as a mapping step followed by a two-dimensional reduction. The original formulation executed the reductions together with the mapping operation, which created too many loop-carried dependencies and resulted in a structurally underperforming design. We, therefore, split the mapping and the two reductions into separate loops and used loop unrolling to improve the loop's throughput at the cost of higher resource usage.

Lastly, we also dealt with issues regarding random number generation: We used uniform random number generators (URNGs) and distribution algorithms from the C++ standard library and Intel's oneDPL to reduce development time. While these implementations work correctly on FPGAs, they were originally designed for CPUs. We especially faced issues with our URNG, the Minstd URNG [PM88], since it uses 64-bit integer multiply and remainder operations. These operations have a high resource usage and latency, and since their output is supposed to appear random, they create a data dependency that leads to an increased initiation interval (II) for the loop that uses them. Therefore, we isolated the random draws into a separate kernel and reduced the II of its main loop to a level below the current bottleneck.

The previous optimizations resulted in a macro-pipeline where different components work independently from one another to execute a chain step. This pipeline needs to be filled at all

times to maximize the mean throughput. We did this by filling the pipeline with initial chain states once, feeding the emitted states back after a chain step, and replacing them with new initial states once the chains are complete. Our scheme ensures that the pipeline is always filled and also simplifies and minimizes interactions with off-chip memory, which additionally reduces compilation times and potentially increases energy efficiency.

The resulting chip design utilizes up to 69% of the resources available on our target device, a Bittware 520N card with an Intel Stratix 10 GX 2800 FPGA. Our design achieves a constant throughput of 577.8 thousand chain steps per second and is bottlenecked by the mapping step of the likelihood computation, as we have predicted based on the synthesis report. Compared to the reference implementation run on one of Noctua 2's compute nodes, our implementation has up to 8.6 times higher throughput. However, our implementation likely has a lower throughput than the one by Ernst et al. [EGJW20], which is an optimized version of the CPU reference implementation.

Our last goal was to verify that our implementation produces results of similar quality compared to the reference implementation. Since the SCITE algorithm is not deterministic and our optimizations did not allow us to implement a bit-perfect copy of the reference, we had to use a statistical test to verify our solution quality. We used the "Two One-Sided t-Tests" (TOST) procedure [Sch87] to define an equivalence test that compares two implementations of the SCITE algorithm, and our implementation is equivalent to the reference implementation with a significance level of 2%.

5.1 Open directions

Time is always a limiting factor during a bachelor's thesis. We therefore tried to limit the scope of the thesis as much as necessary to assure that we meet our goals. This however means that there are some optional or alternative features of the original SCITE implementation we were unable to implement. Additionally, it is almost always possible to further optimize an application. In this section, we therefore discuss both the removed features and possibilities for future optimizations that we saw during this thesis.

Feature: Co-optimal trees The most practical feature of the reference implementation that we could not implement is exporting co-optimal trees. For most inputs, there are multiple trees with the same, maximum likelihood. A user will probably want to know about all of these trees in order to interpret the results correctly. However, there is a high probability that these trees are visited multiple times, and an implementation of the SCITE algorithm therefore needs to check for duplicates. This is not a trivial task, especially on FPGAs. We therefore initially decided to only store one maximum likelihood tree at first and then extend our design once the rest is satisfactory. However, we did not reach this point and it therefore remains an open direction.

Feature: β search There is one feature that we actually implemented, but that we did not mention in this thesis: The reference implementation also supports searching for the most likely false-negative probability β by performing a gaussian walk as an additional move. We simply copied the code that is relevant to this feature and it did not cause any performance issues. Since we did not change it, we also did not need to verify it, and since none of our work is related to the β search, we decided to omit it from the thesis entirely. We should also mention that SCITE and ffSCITE do not perform a β search for default parameters, and we did not use this feature during benchmarking.

Feature: Additional scoring methods The first truly optional features we chose not to implement are additional scoring methods for mutation trees. The first one is the so-called "sum score" which was published as a part of the reference implementation of SCITE. It does not compute the likelihood of a true mutation matrix, but the a posteriori probability of the true mutation matrix given the input mutation matrix. In the language of Bayesian probability theory, it computes $\mathbb{P}(E|D)$ instead of $\mathbb{P}(D|E)$ where D is the noisy mutation matrix and E is the true mutation matrix to score. The main technical difference is that it additionally potentiates the scores of the individual cell-node pairs, executes a sum-reduction on all possible attachment points of a cell, and then takes the logarithm of the sum for the global sum reduction. We assume that an implementation of this a posteriori score would require even more hardware or have less throughput than the likelihood score since its computation is already the most-unrolled and most resource-consuming part of the design.

The second scoring method is ∞SCITE, which was published by Kuipers et al. [KJRB17] as a successor to SCITE to test the infinite sites assumption. We initially set ourselves the optional goal to at least evaluate whether an adaption of ffSCITE to the ∞SCITE scoring scheme is possible, but we were not able to do this in time for this thesis.

Feature: Homo- and heterozygous mutations Another unimplemented feature is the support for homozygous and heterozygous mutations. As far as we understand it, homozygous mutations are mutations where both chromosomes carry the mutation, and heterozygous mutations are mutations where only one chromosome carries it. This was implemented to support the data provided by Hou et al. [HSZ+12], which reports these two types of mutations. Since we tried to keep our implementation simple, we chose not to implement it.

Feature: Continous outputs The reference implementation continuously emits information about the current chain and step index as well as the current maximum score. Additionally, it can also write samples from the chains into separate files. These features are very useful to users as they allow them to make predictions of the required number of chains and chain steps as well as the runtime. ffSCITE however does not emit this information since it only writes back its results after the entire execution is complete. Implementing these features does not require much effort, but it was not necessary for our goals and has therefore not been done yet.

Optimization: Customized random draws The current scheme leaves much room for optimization. First of all, there may be URNGs that work better on FPGAs and have a shorter critical path than Minstd. We think that it is probably impossible to remove the data dependency completely since the output is supposed to appear random, but there may be room for improvement. Secondly, our current approach is very wasteful with random bits: For example, we use Minstd's output, which has up to $m = 2^{32} - 1$ possible values, to sample one of 64 nodes and then draw another number to sample one of the other 63 nodes. Since we have upper bounds for all set sizes we want to sample from, it might be possible to sample multiple move parameters from one URNG draw and save resources and execution time. However, we assume that this will require much more research outside our current expertise and time to implement and verify.

Optimization: Resolve compiler issues with bigger inputs ffSCITE currently supports inputs with up to 64 cells and 63 genes. Compiling the design with more cells and genes however leads to internal compiler errors that we were not able to resolve. If ffSCITE is supposed to be used by practitioners, then this issue needs to be resolved since real inputs are likely to be bigger than 64x63.

Optimization: Replicate the design The current design uses only one replication of the pipeline. If the resource usage can be reduced, it may be possible to fit the same design twice on one Intel Stratix 10 FPGA so that the outputs of the first replication are fed into the second replication and both work in parallel. Additionally, we have the option at the Noctua 2 supercomputer to expand the design to multiple interconnected FPGAs which can then work in parallel too. Ernst et al. [EGJW20] have shown that the SCITE algorithm can easily be parallelized on independent cores or devices, so this should also be possible to accomplish with FPGAs too.

Exploration: Energy Efficiency FPGAs are known by some for their energy efficiency [BTL10] since developers have direct control over data movement and data subsequently needs to be moved less inside the FPGA and between the FPGA and external memory. This may be true especially in the case of SCITE since the reference implementation always writes the resulting state back to a relatively big buffer which may or may not reside in a cache, while ffSCITE never copies intermediate states to the global memory in the first place. We, therefore, assume that ffSCITE is more energy efficient than the CPU-based versions by Jahn et al. [JKB16] and Ernst et al. [EGJW20]. However, we would need to verify this and it is therefore an open direction.

Bibliography

[BTL10] Brahim Betkaoui, David B Thomas, and Wayne Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101. IEEE, 2010.

[EGJW20] Dominik Ernst, Peter Georg, Katharina Jahn, and Tilo Wettig. Performance engineering for infscite. 2020.

[GM12] Mel Greaves and Carlo C Maley. Clonal evolution in cancer. *Nature*, 481(7381):306–313, 2012.

[GVG12] Robert J Gillies, Daniel Verduzco, and Robert A Gatenby. Evolutionary dynamics of carcinogenesis and why targeted therapy does not work. *Nature Reviews Cancer*, 12(7):487–493, 2012.

[HSZ+12] Yong Hou, Luting Song, Ping Zhu, Bo Zhang, Ye Tao, Xun Xu, Fuqiang Li, Kui Wu, Jie Liang, Di Shao, et al. Single-cell exome sequencing and monoclonal evolution of a jak2-negative myeloproliferative neoplasm. *Cell*, 148(5):873–885, 2012.

[JKB16] Katharina Jahn, Jack Kuipers, and Niko Beerenwinkel. Tree inference for single-cell data. *Genome Biology*, 17(86), 2016.

[KJRB17] Jack Kuipers, Katharina Jahn, Benjamin J Raphael, and Niko Beerenwinkel. Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors. *Genome research*, 27(11):1885–1894, 2017.

[KMN18] R. Kastner, J. Matai, and S. Neuendorffer. Parallel Programming for FPGAs. *ArXiv e-prints*, May 2018.

[Leh51] Derrick H Lehmer. Mathematical methods in large-scale computing units. *Annu. Comput. Lab. Harvard Univ.*, 26:141–146, 1951.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[Nav14] Nicholas E Navin. Cancer genomics: one cell at a time. *Genome biology*, 15(8):1–13, 2014.

[NZVLW+12] Serena Nik-Zainal, Peter Van Loo, David C Wedge, Ludmil B Alexandrov, Christopher D Greenman, King Wai Lau, Keiran Raine, David Jones, John Marshall, Manasa Ramakrishna, et al. The life history of 21 breast cancers. *Cell*, 149(5):994–1007, 2012.

[oDCC22] Global Burden of Disease 2019 Cancer Collaboration. Cancer Incidence, Mortality, Years of Life Lost, Years Lived With Disability, and Disability-Adjusted Life Years for 29 Cancer Groups From 2010 to 2019: A Systematic Analysis for the Global Burden of Disease Study 2019. *JAMA Oncology*, 8(3):420–444, 03 2022.

[PM88] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[PMS93] Stephen K. Park, Keith W. Miller, and Paul K. Stockmeyer. Remarks on choosing and implementing random number generators. Technical Correspondence in *Communcations of the ACM*, 36(7):108-110, 1993.

[RAMV07] Juan J Rodriguez-Andina, Maria J Moure, and Maria D Valdes. Features, design tools, and application domains of fpgas. *IEEE Transactions on Industrial Electronics*, 54(4):1810–1823, 2007.

[SCF09] Michael R Stratton, Peter J Campbell, and P Andrew Futreal. The cancer genome. *Nature*, 458(7239):719–724, 2009.

[Sch87] Donald J Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of pharmacokinetics and biopharmaceutics*, 15(6):657–680, 1987.

[Sna00] Steven M Snapinn. Noninferiority trials. *Trials*, 1(1):1–3, 2000.

[Swa12] Charles Swanton. Intratumor heterogeneity: evolution through space and time. *Cancer research*, 72(19):4875–4882, 2012.

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Eidesstattliche Versicherung

Nachname: Opdenhövel Vorname: Jan-Oliver

Matrikelnr.: 7151509 Studiengang: Informatik Bachelor of Sciences

(•) Bachelorarbeit ( ) Masterarbeit

Titel der Arbeit: Accelerating Single Cell Inference of Tumor Evolution with FPGAs

☐ Die elektronische Fassung ist der Abschlussarbeit beigefügt.

☒ Die elektronische Fassung sende ich an die/den erste/n Prüfenden bzw. habe ich an die/den erste/n Prüfenden gesendet.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit (Ausarbeitung inkl. Tabellen, Zeichnungen, etc.) selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Abschlussarbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung entspricht der gedruckten und gebundenen Fassung.

Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist die Vizepräsidentin / der Vizepräsident für Wirtschafts- und Personalverwaltung der Universität Paderborn. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz NRW in der aktuellen Fassung).

Die Universität Paderborn wird ggf. eine elektronische Überprüfung der Abschlussarbeit durchführen, um eine Täuschung festzustellen.

Ich habe die oben genannten Belehrungen gelesen und verstanden und bestätige dieses mit meiner Unterschrift.

Ort: Paderborn Datum: 22.09.2022

Unterschrift: _____

Datenschutzhinweis:

*Die o.g. Daten werden aufgrund der geltenden Prüfungsordnung (Paragraph zur Abschlussarbeit) i.V.m. § 63 Abs. 5 Hochschulgesetz NRW erhoben. Auf Grundlage der übermittelten Daten (Name, Vorname, Matrikelnummer, Studiengang, Art und Thema der Abschlussarbeit) wird bei Plagiaten bzw. Täuschung der*die Prüfende und der Prüfungsausschuss Ihres Studienganges über Konsequenzen gemäß Prüfungsordnung i.V.m. Hochschulgesetz NRW entscheiden. Die Daten werden nach Abschluss des Prüfungsverfahrens gelöscht. Eine Weiterleitung der Daten kann an die*den Prüfende*n und den Prüfungsausschuss erfolgen. Falls der Prüfungsausschuss entscheidet, eine Geldbuße zu verhängen, werden die Daten an die Vizepräsidentin für Wirtschafts- und Personalverwaltung weitergeleitet. Verantwortlich für die Verarbeitung im regulären Verfahren ist der Prüfungsausschuss Ihres Studiengangs der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.*