Mutation Tree Reconstruction of Tumor Cells on FPGAs Using a Bit-Level Matrix Representation

Jan-Oliver Opdenhövel
Paderborn University, Paderborn
Center for Parallel Computing
Paderborn, Germany
joo@mail.uni-paderborn.de

Christian Plessl
Paderborn University, Paderborn
Center for Parallel Computing
Paderborn, Germany
plessl@uni-paderborn.de

Tobias Kenter
Paderborn University, Paderborn
Center for Parallel Computing
Paderborn, Germany
kenter@uni-paderborn.de

ABSTRACT

Reconstructing the mutation history of a cancer cell in the form of a phylogenetic tree from noisy genome sequencing data requires a likelihood model and a search or optimization strategy. The “Single Cell Inference of Tumor Evolution” (SCITE) software performs such a search via Monte Carlo Markov Chains that explore and evaluate different random update operations on a current candidate tree. In this work, we present an FPGA accelerator design (denoted as “bit-based SCITE” (bbSCITE)) that implements all required operations with deterministic latency as bit-operations on a matrix representation. The largest design updates and counts 36100 set bits in each clock cycle and thus outperforms the original single-threaded software reference by up to 80×.

CCS CONCEPTS

• **Computer systems organization → Reconfigurable computing**; • **Applied computing → Bioinformatics**; • **Hardware →** High-level and register-transfer level synthesis.

KEYWORDS

FPGA, Monte Carlo Markov Chain (MCMC), Bit-level parallelism, SYCL, oneAPI

ACM Reference Format:

Jan-Oliver Opdenhövel, Christian Plessl, and Tobias Kenter. 2023. Mutation Tree Reconstruction of Tumor Cells on FPGAs Using a Bit-Level Matrix Representation. In *The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies 2023 (HEART 2023), June 14–16, 2023, Kusatsu, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3597031.3597050

1 INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) have been used with great success for the acceleration of workloads from bioinformatics. A particular focus has been on sequence alignment [2, 5, 6], where datatypes with few bits for nucleotides or amino acids allowed to create highly parallel customized datapaths. Rapid advancements in genome sequencing technology and basic analysis techniques as the identification of genome differences in the Single-Nucleotide Polymorphism (SNP) are a foundation for ever more sophisticated analysis procedures on top of them. One such direction is reconstructing the mutation history of cancer cells based on single-cell sequencing.

The software approach published as “Single Cell Inference of Tumor Evolution” (SCITE) [3] is by itself again a computationally demanding workload that can profit from FPGA acceleration. However, the involved algorithms and data structures are different from the foundational components of sequence alignment. The SCITE software reference [3] maintains and modifies a tree structure representing a hypothesis for the mutation history that is regularly transformed into a matrix structure to calculate the likelihood of the current hypothesis. As the involved tree traversals and modifications generally involve a dynamic number of sequential sub-steps, it is inherently challenging to create a well-utilized computation pipeline with parallelism for them.

In this work, we present the first FPGA accelerator for SCITE and resolve the challenge of tree data structures by reformulating the tree updates as parallel binary operations on equivalent bit-matrix representations. Our further contributions include:

- A meta-pipeline that overlaps the evolution of several independent Monte Carlo Markov Chains (MCMCs) to obtain full utilization of the datapath.
- Support for different problem sizes at runtime with design parameterization at compile time.
- End-to-end speedups of up to 80× against a sequential CPU reference and a performance model that explains the measured throughput.
- A comparison with an idealized CPU performance proxy for the throughput-critical likelihood calculation step.

2 APPLICATION BACKGROUND

SCITE [3] is an algorithm and code from the domain of bioinformatics. It solves the real-world problem where multiple cells are extracted from a patient’s tumor and a practitioner wants to analyze which mutations are present in the tumor and lead to its current state. There are single-cell sequencers available that can analyze the genome of a single cell and report whether certain genes are in their common, unmutated form or whether they are mutated. However, these results are often noisy. Sometimes, the sequencers report a mutation that is not present, sometimes they fail to find a mutation, and sometimes the gene is lost in the process and therefore no statement about its mutation status can be made. The sequencing output is represented as a matrix $D[cell][gene]$ with one row for every cell that is analyzed and one column for every gene that is considered. The matrix entries are either 0 for no mutation, 1 for a mutation,

HEART 2023, June 14–16, 2023, Kusatsu, Japan
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0043-9/23/06.
https://doi.org/10.1145/3597031.3597050

and 2 for missing data. As stated before, this reported mutation matrix D contains errors due to the sequencing process and often does not represent a biologically plausible mutation history of the analyzed cells. We define E as the true mutation matrix for these cells. The SCITE approach to reconstruct E builds on the one hand on a likelihood model presented in the next paragraph that quantifies how well a candidate for E fits to the observations encoded in D, and on the other hand on a mutation model that defines a search space of possible candidates for E, described afterwards.

2.1 Likelihood Model

For the likelihood model, we assume that the errors the sequencer makes for every entry of D are independent of another. If the data for an entry is not missing, we assume that there is a probability for false positives called $\alpha \in [0,1]$ and a probability for false negatives called $\beta \in [0,1]$. Conversely, the probability for a true positive is $1-\beta$ and the probability for a true negative is $1-\alpha$. Formally, this means that we have the following for every cell c and gene g:

$$\mathbb{P}(D[c][g] = 1 \mid E[c][g] = 0 \land D[c][g] \neq 2) = \alpha \tag{1}$$
$$\mathbb{P}(D[c][g] = 0 \mid E[c][g] = 0 \land D[c][g] \neq 2) = 1-\alpha \tag{2}$$
$$\mathbb{P}(D[c][g] = 0 \mid E[c][g] = 1 \land D[c][g] \neq 2) = \beta \tag{3}$$
$$\mathbb{P}(D[c][g] = 1 \mid E[c][g] = 1 \land D[c][g] \neq 2) = 1-\beta \tag{4}$$

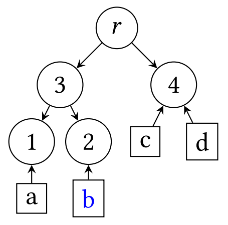
With this probability model, we can build a likelihood function that measures the likelihood of a given candidate for E by multiplying the probabilities for every matrix entry:

$$\Lambda(E,D) := \prod_{c\in C}\prod_{g\in G}\begin{cases}1-\alpha & D[c][g] = 0 \land E[c][g] = 0\\ \beta & D[c][g] = 0 \land E[c][g] = 1\\ \alpha & D[c][g] = 1 \land E[c][g] = 0\\ 1-\beta & D[c][g] = 1 \land E[c][g] = 1\\ 1 & D[c][g] = 2\end{cases} \tag{5}$$

The task of SCITE is to find the most likely true mutation matrix E for a given input mutation matrix D.

2.2 Mutation Tree Model

In order to define a search space for plausible mutation matrices, the so-called "infinite sites assumption" is made in SCITE. It assumes that every gene mutates exactly once in the history of the tumor and that the first cell with such a mutation passes it down to all of its descendants that are created via cell division. This assumption allows to model the mutation history as a tree. Every node represents a mutation state and is labeled with a gene. The root r represents the initial, unmutated state of the genome and every child adds a mutation of its label gene to its parent's state. In other words, a gene g is mutated in the state of a node v iff there is a node w on the path from r to v with the label g. During the cell-extraction process, samples from any node in the mutation tree can be encountered, which is modeled by attaching cells to tree nodes. Fig. 1 gives an example how of a mutation tree with respective E and D may look like. During the sequencing stage, errors are introduced from E to D, and during the reconstruction phase, we look for an E that has the highest likelihood w.r.t. to D and at the same time can be modeled with a mutation history that forms a tree. While E is only completely defined by the combination



E	1	2	3	4
a	1	0	1	0
b	0	1	1	0
c	0	0	0	1
d	0	0	0	1

D	1	2	3	4
a	1	0	1	0
b	0	2	1	0
c	1	0	2	1
d	0	2	0	1

Figure 1: An example of a mutation tree, and related mutation matrices E and D. The node r represents the initial, unmutated state, the round nodes with numeric labels represent mutations of individual genes in the tree, and the square nodes with letter labels represent cells that are attached to the tree nodes. The true mutation matrix E and the noisy mutation matrix D have one row for each cell and one column for each gene. Exemplarily, the highlighted cell b contains mutations in genes 2 and 3 (second row of E), but during readout, the information on gene 2 was lost (entry 2 in second row of D).

of mutation tree and cell attachment, SCITE actually explores the candidate space for mutation trees and calculates the likelihood of each tree by identifying the most likely attachment for each cell in each evaluation step.

2.3 Search Procedure

The search procedure to explore candidate mutation trees is a Monte Carlo Markov Chain (MCMC). It starts with a randomly generated tree and iteratively applies random modifications to it. If the resulting tree, together with the most likely attachments, is more likely to be correct than the previous tree, it is accepted as the new solution. If the new tree is less likely to be correct, a random threshold between 0 and 1 is drawn. If the ratio of the proposed likelihood to the current likelihood is lower than this threshold, the modification is rejected. This kernel loop of modifying the tree, computing its likelihood, and deciding on the new solution is then repeated multiple times and also restarted with new trees. For statistical details on the convergence of this search procedure, we refer to Jahn et al. [3].

3 FPGA DESIGN APPROACH

In this section, we present the FPGA design denoted as "bit-based SCITE" (bbSCITE) in terms of overall kernel structure, data representation, updates of mutation tree candidates, and parallel likelihood calculation. Although written with a specific implementation in mind, these design aspects are largely independent of the target FPGA architecture and used tool chain.

3.1 Kernel Structure

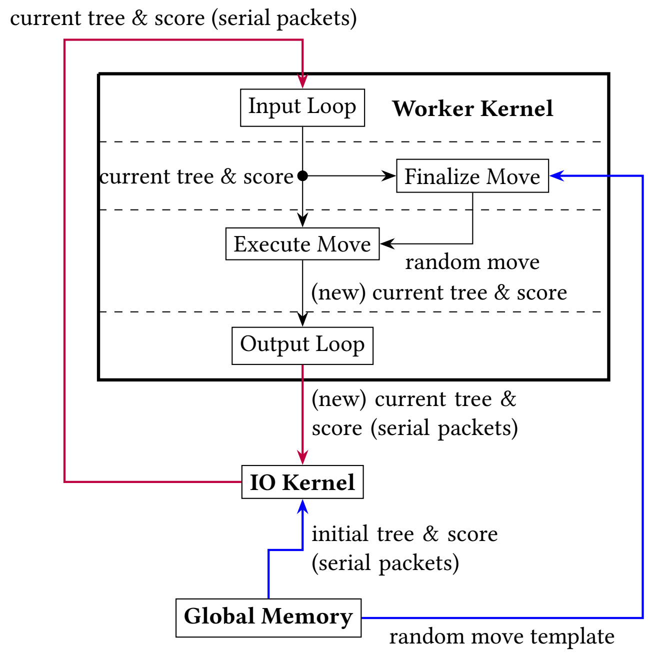
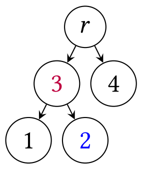

current tree & score (serial packets)
Input Loop
Worker Kernel
current tree & score
Finalize Move
Execute Move
random move
(new) current tree & score
Output Loop
(new) current tree & score (serial packets)
IO Kernel
initial tree & score (serial packets)
Global Memory
random move template

Figure 2: Block diagram of the FPGA design of bbSCITE. The IO kernel and the worker kernel are connected via pipes, colored in violet, and the kernels access the global memory via load-store units, colored in blue. The sections separated by the dashed lines inside the worker kernel run in parallel and use/forward the values from the previous stage.

The general FPGA design that realizes the MCMC algorithm is illustrated in Fig. 2 and contains a worker kernel and an IO kernel. These kernels are circularly connected using pipes realized as on-chip FIFO buffers. The IO kernel sends the current tree and its score to the worker kernel and with a design-specific latency, the worker kernel sends the result back to the IO kernel. Each transfer is done sequentially in several packets since a pipe with a sufficient bitwidth to transfer a full tree in one cycle would use excessive routing resources and would not be utilized in each cycle anyway. The worker kernel receives the current tree and its score with an input loop. Then, the kernel decides how to modify the tree. In order to save resources for a random number generator on the FPGA and since little off-chip bandwidth is used by the rest of the design, random move templates are pre-computed on the host. Before kernel invocation, they are transferred to the global memory of the accelerator card and streamed in and finalized with the knowledge of the current tree inside the worker kernel. Once the suggested move is decided, the "Execute Move" step executes the move, computes the likelihood score of the proposed tree, and decides whether the proposed tree is accepted as the new tree. As stated earlier, a proposed tree is accepted if it has a higher likelihood than the current tree or if the ratio of the proposed and the current likelihood is above a random threshold. This threshold is also fetched from global memory as part of the move template. The updated tree is fed back to the IO kernel.

These four parts, the input loop, the move finalization, the tree modification and scoring, and the output loop can work independently and form a meta-pipeline. The worker kernel can therefore process multiple chains at once, which is exploited by the IO kernel. It assumes a certain capacity for the worker kernel and initially feeds a corresponding number of chains into the worker kernel. Then, the previous outputs of the worker kernel are fed back into the loop, which maintains the occupation of the worker kernel despite relatively high latency of individual steps. Once the requested chain steps were executed, the output states are discarded and if more initial states are available, those are fed into the worker kernel instead of the previous output. Otherwise, the worker kernel is flushed and execution halts. The worker kernel also keeps track of the most likely candidate tree encountered so far and writes it back to global memory once the execution is finished.

3.2 Ancestor and descendant matrices



A	1	2	3	4	r
1	1	0	0	0	0
2	0	1	0	0	0
3	1	1	1	0	0
4	0	0	0	1	0
r	1	1	1	1	1

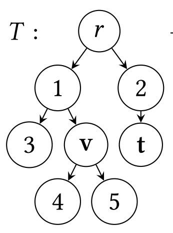
A^T	1	2	3	4	r
1	1	0	1	0	1
2	0	1	1	0	1
3	0	0	1	0	1
4	0	0	0	1	1
r	0	0	0	0	1

Figure 3: The mutation tree from Fig. 1 with the corresponding ancestor matrix A and the descendant matrix A^T. The highlighted column 2 in the ancestor matrix contains an entry 1 for all nodes on the path from the root r to the given node 2, i.e. $A[r][2] = A[3][2] = A[2][2] = 1$. Its transpose in A^T is related to the mutation matrix E in Fig. 1: The first two rows are identical except for the inclusion of the root node since the cells a and b are attached to nodes 1 and 2, respectively. However, the fourth row of A^T appears in the third and fourth row of E since both c and d are attached to node 4, and the third row of A^T does not appear in E since no cell is attached to node 3.

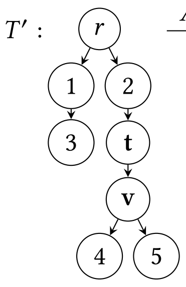
The main innovation in bbSCITE is the use of an ancestor matrix as the canonical tree data structure throughout the entire design. An ancestor matrix A is a matrix of binary values with a row and a column for every node in the tree. For every pair of nodes (v, w), the matrix contains a 1 if and only if there is a path in the tree from v to w, i.e. v is an ancestor of w (including w itself). Otherwise, it contains a 0. Conversely, a descendant matrix is a matrix that has a 1 at position (v, w) iff v is a descendant of w; It is the transpose of an ancestor matrix. An example is given in Fig. 3, depicting a tree on top along with the corresponding ancestor matrix A and descendant matrix A^T below it. This data structure makes it easy to check whether a cell c, attached to a node v, is mutated at gene g: g is mutated if and only if g is an ancestor of v, which is indicated by $A[g][v] = 1$. We have implemented this data structure as a 1d-array of bit-vectors. One bit-vector represents one row of the ancestor matrix and each bit in it represents one matrix entry. Looking up the matrix element $A[v][w]$, therefore, involves loading the v-th bit-vector and isolating the w-th bit. On the other hand, all operations that need to be executed in parallel for an entire matrix row can be expressed as word level operations on the bit-vectors, allowing for a compact code representation that is translated into bit-parallel hardware blocks.

Jahn et al. [3] have already used an ancestor matrix in their reference implementation to compute a tree's likelihood, but their canonical data structure is a parent vector. This is a list of node indices where the i-th element contains the index of the parent of node i. The parent vector allows for easy modifications of the tree structure with only a few updates to parent indices but always requires a reconstruction of the ancestor matrix for the subsequent likelihood calculation. Tree updates and translating the parent vector structure into a bit-matrix both require variable numbers of steps and are thus hard to pipeline on FPGAs. Consequently, we instead use ancestor matrices throughout the entire design and also implemented the tree modifications on them to obtain a bit-parallel and predictable FPGAs pipeline.

3.3 Mutation tree updates



A	1	2	3	4	5	v	t	r
1	1	0	1	**1**	**1**	**1**	0	0
2	0	1	0	**0**	**0**	**0**	1	0
3	0	0	1	0	0	0	0	0
4	0	0	0	1	0	0	0	0
5	0	0	0	0	1	0	0	0
v	0	0	0	1	1	1	0	0
t	0	0	0	**0**	**0**	**0**	1	0
r	1	1	1	1	1	1	1	1



A'	1	2	3	4	5	v	t	r
1	1	0	1	**0**	**0**	**0**	0	0
2	0	1	0	**1**	**1**	**1**	1	0
3	0	0	1	0	0	0	0	0
4	0	0	0	1	0	0	0	0
5	0	0	0	0	1	0	0	0
v	0	0	0	1	1	1	0	0
t	0	0	0	**1**	**1**	**1**	1	0
r	1	1	1	1	1	1	1	1

Figure 4: Mutation tree and ancestor matrix representations before and after exemplary "Prune and Reattach" move where node v is pruned from node 1 and reattached to node t. Changes between A and A' marked in bold, colors highlight partial rules to define A'.

Algorithm 1 Given a tree with nodes in V, represented as an ancestor matrix $A \in \{0,1\}^{|V|\times|V|}$, and nodes v and t, compute the ancestor matrix A' of the tree after the "Prune and Reattach" move with the parameters v and t.

```
1:  $A' \leftarrow 0 \in \{0,1\}^{|V|\times|V|}$
2:  for all $x \in V$ do
3:      $Axt \leftarrow A[x][t]$ ▷ Extract bit, 1 iff $x$ is ancestor of $t$
4:      $Avx \leftarrow A[v][x]$ ▷ Extract bit, 1 iff $v$ is ancestor of $x$
5:      for all $y \in V$ do ▷ Parallel on complete bit-vector
6:          if $A[v][y]$ then
7:              ▷ Modify columns where $v$ is ancestor of $y$:
8:              ▷ Connect via $t$ or retain subtree below $v$
9:              $A'[x][y] \leftarrow Axt \vee (Avx \wedge A[x][y])$
10:         else
11:             ▷ Keep other columns
12:             $A'[x][y] \leftarrow A[x][y]$
13:         end if
14:     end for
15: end for
16: return $A'$
```

In this section, we introduce the "Prune and Reattach" tree move as an example of the tree modification algorithms on ancestor matrices that we have developed for bbSCITE. The other moves are similar in nature. For this move, two nodes v and t are sampled randomly. v is sampled uniformly from all non-root nodes, but t is sampled uniformly from all non-descendants of v so that there is no path from v to t in the tree. Then, the node v is moved from its current parent to node t. In other words, the entire subtree below v is moved to a new parent. If there were a path from v to t, the move would introduce a circle in the tree, which is why we sample t from all non-descendants of v. Fig. 4 contains an example for this move, as well as the ancestor matrices of the tree before and after the move. In a parent vector encoding (not shown), only a single index would be changed, but in the ancestor matrix representation, multiple bits are different.

Given the inputs A and the move parameters v and t, the goal is to calculate A' with simple bit-level operations. Alg. 1 presents our approach to build the new ancestor matrix element by element. For every pair of nodes x and y, we find out whether there will be a path from x to y in the new tree. The answer to this question is the entry in the ancestor matrix for this pair. In the tree notation, the edge from the old parent of v to v is removed and a new edge from t to v is added. This means that the connectivity of x and y is only changed if the path between them would contain v. If there is no path from v to y, then the potential path between x and y can not contain v. In this case, nothing changes and the entry from the old ancestor matrix is copied to the new one (Alg. 1, lines 6, 12). However, if there is a path from v to y, then there are only two cases where a path from x to y can exist in the new tree: Either there is a path from x to t, or the entire path between x and y is inside the subtree below v. If there is a path from x to t, then we can construct a path from x to y via the newly introduced edge from t to v, and if the path from x to y is entirely inside the subtree below v, then it is undisturbed by the move. The boolean expression

$Axt \lor (Avx \land A[x][y])$ (Alg. 1, line 9) covers exactly these cases, implicitly setting detached connections (in Fig. 4 the subtree below node 1) and unrelated nodes (in Fig. 4 node 3) to 0.

Ancestor matrices are implemented as a list of bit vectors. The first index denotes a word in a RAM block and the second index denotes a bit in this vector. If we therefore fully unroll the inner loop over all nodes y, we get a single logic block that works on all bits of the vector in parallel. The hardware implementation first loads $A[v]$ to keep it in a register and then runs a single pipelined loop that loads $A[x]$, passes it through the logic, and stores the result in $A'[x]$. There are no loop-carried dependencies and the loop iteration space is fixed, unlike the ancestor matrix construction that this algorithm replaces. The other tree moves are implemented in the same way with a single pipelined loop with $|V|$ iterations and parallel logic operations per bit-vector word in the matrix.

3.4 Likelihood computation

The same approach with parallel binary operations over bit-vectors and a pipelined loop over rows of the binary matrix is also applied to the likelihood calculation. Since the original mutation matrix $D \in \{0, 1, 2\}^{|\text{Cells}|\times|\text{Genes}|}$ is not in a binary format, we instead encode it with two binary matrices `is_known` and `is_mutated` $\in \{0, 1\}^{|\text{Cells}|\times|V|}$. We define `is_known`$[c][g] := (D[c][g] \neq 2)$ to encode whether the mutation status of a cell-gene combination is known, and define `is_mutated`$[c][g] := (D[c][g] = 1)$ to encode the actual mutation status. These matrices are extended by one column such that the bit-vector format matches the one of the ancestor matrices. The `is_known` entries for this extra column are 0, such that it does not impact the likelihood value. Using these, calculating the likelihood of an attachment of a cell c to a mutation tree node v is possible with bit-wise logic operations. Looking at individual bits, we have a true positive mutation at gene g if

$$\texttt{is\_known}[c][g] \land \texttt{is\_mutated}[c][g] \land A^T[v][g] \tag{6}$$

is true, which means that the cell-gene combination is reported as mutated and the underlying mutation tree model also reports the combination as mutated. Encoded as a complete bit-vector with word-level binary operations, the vector

$$\texttt{is\_known}[c] \land \texttt{is\_mutated}[c] \land A^T[v] \tag{7}$$

contains a 1 bit for every true positive that has occurred for this cell and these occurrences can be counted with the so-called popcount function. Conversely, using the inverse $\neg$`is_mutated`$[c]$ vector allows to find and count false positives, and using the inverse $\neg \mathrm{A}^\mathrm{T}[\mathrm{v}]$ for false respectively true negatives. The thus counted number of event occurrences is then multiplied by the respective logarithmic probability of the event and summed up to receive the log-likelihood of the entire cell attachment.

Alg. 2 utilizes this way of computing the log-likelihood. The log-likelihood has the same ordering properties as the likelihood, but allows to replace exponentiation with multiplication (Alg. 2, line 33). In order to find the most likely attachment of each cell to a node, the likelihood calculation of SCITE as well as Alg. 2 originally contain three nested loops, one over cells, one over possible attachments (i.e. nodes in A^T) and one over genes per cell or node. Following the vector-level notation of Eq. 7, Alg. 2 replaces the loop over genes by bit-vectors that automatically yield parallel operations. On the other hand, to explicitly cover the four possible event classes with their respective `probability[posterior][prior]` values, Alg. 2 contains an explicit loop over all possible events (line 11). In order to generate parallel operations for these four classes, the loop is annotated with the unrolling attribute and will be specialized at compile time to the specific cases. To balance the throughput of the likelihood calculation with that of the other steps, we also mark

Algorithm 2 Given a tree with nodes in V, represented as an descendant matrix $A^T \in \{0, 1\}^{|V|\times|V|}$, the mutation data matrices is_known, is_mutated $\in \{0, 1\}^{|\text{Cells}|\times|V|}$, and the error probabilities $\alpha, \beta \in [0, 1]$, compute the likelihood function from Subsection 3.4, as used in bbSCITE.

```
1:  tree_score ← 0
2:  probability ← [[log(1 − α), log(β)], [log(α), log(1 − β)]]
3:
4:  for all c ∈ Cells do ▷ Pipelined sequentialy
5:      max_cell_score ← −∞
6:
7:      for all v ∈ V do ▷ Parallel, i.e. unrolled completely
8:          cell_score ← 0
9:
10:         ▷ Cover all cases, unrolled completely
11:         for all prior, posterior ∈ {0, 1} × {0, 1} do
12:             ▷ Bit-vector load of known occurrences
13:             v_occurrences ← is_known[c]
14:
15:             ▷ Load and AND bit-vector of predicted cases
16:             if posterior = 1 then
17:                 v_posterior ← A^T[v]
18:             else
19:                 v_posterior ← ¬A^T[v]
20:             end if
21:             v_occurrences ← v_occurrences ∧ v_posterior
22:
23:             ▷ Load and AND bit-vector of reported cases
24:             if prior = 1 then
25:                 v_prior ← is_mutated[c]
26:             else
27:                 v_prior ← ¬is_mutated[c]
28:             end if
29:             v_occurrences ← v_occurrences ∧ v_prior
30:
31:             ▷ Compute and add score for this case
32:             p ← probability[posterior][prior]
33:             case_score ← p · popcount(v_occurrences)
34:             cell_score ← cell_score + case_score
35:         end for
36:
37:         if cell_score > max_cell_score then
38:             max_cell_score ← cell_score
39:         end if
40:     end for
41:     tree_score ← tree_score + max_cell_score
42: end for
43: return tree_score
```

the next loop dimension (line 7) with an unrolling attribute, which reduces the sequential steps of this function from cubic to linear, with a quadratically scaling bit-parallel datapath, which processes $4 * |V|^2$ bits per clock cycle.

4 IMPLEMENTATION AND EVALUATION

bbSCITE is implemented as a high-level synthesis (HLS) design written in C++20, using SYCL and Intel OneAPI version 22.3.0. The target FPGA is a Intel Stratix 10 GX 2800 FPGA on a Nallatech 520N PCIe card. The utilized backend tool chain is Intel Quartus Pro version 20.4.0 Build 72, matching the version of the board support package (*20.4.0_hpc*) that provides PCIe and DDR memory interfaces.

4.1 Hardware synthesis

All components inside the worker kernel (Sec. 3.1) are implemented as pipelined loops with an initiation interval (II) of 1 that iterate over the tree nodes. The maximal number of nodes in a tree is fixed as a design parameter during compilation. This means that the number of bits in a bit-vector, the number of vectors in an ancestor matrix, and therefore also the number of iterations and the width of the data path are fixed with this parameter. The biggest vector size that we have found to be reliably synthesizable for this target is 96 bits; The resource usage for such a design instance as well as for a smaller 64-bit variant is listed in Tab. 1. Both designs are relatively logic-intensive, on the one hand due to many bit-level operations, mostly in the "Execute Move" loop, and on the other hand to orchestrate movement of the bit-matrices. The floating-point digital signal processors (DSPs) of the Stratix 10 architecture see some usage for the arithmetic parts of the likelihood calculation, and not many on-chip RAM blocks are used. Despite different resource usage, both designs achieve similar clock frequencies around 310 MHz.

The bottleneck for larger vector sizes is routing. When trying to compile the design with 128-bit vectors with an unlimited clock speed, the demand for interconnect wires peaks up to 125%. This is only eliminated by lowering the clock speed to the regions around 150 MHz, which is less than a third the maximum clock speed of 480 MHz for the Intel Stratix 10 FPGA and less than half the clock speeds of the smaller design instances. Even with a lowered clock speed, timing errors still occur and we were thus unable to reliably synthesize the design with 128 bits. There also seems to be an issue with the arbitrary precision integer type[1] that we use to implement bit vectors. The last bit of the 96-bit vector is always set to zero and thus limits the usable number of nodes in our experiments to 95. The problem does not show up with 32-bit, 64-bit, or spurious 128-bit design instances.

4.2 Performance benchmark

In order to compare the performance of bbSCITE to the reference SCITE implementation by Jahn et al. [3], we measure the throughput in Markov chain steps per second. For this benchmark, we have generated random input data with different sizes, using the stochastic assumptions presented in Section 2 and arbitrary error probabilities. We wanted to test both bbSCITE variants with fully and partially utilized bit vectors and therefore chose 32 bits, 64 bits, and 95 bits as input sizes. For 32 bits, the input covers 32 cells and 31 genes so that the mutation tree has 32 nodes. Other input sizes are analogous, and we had to use 95 bits as the biggest input size since the 96th bit of 96-bit bbSCITE is faulty (see Subsection 4.1). Then, we simulated 48 Markov chains with 2,000,000 chain steps each for every input and measured the runtime, as well as the power draw of the bbSCITE instances. We repeated this experiment four times and took the average of those repetitions as the final value.

This performance data is shown in Tab. 2, where we have listed the throughput of the three executables in executed Markov chain steps per second. We see that both bbSCITE variants are significantly faster than the CPU-based SCITE implementation, with speedups of up to 80×. We also see that the speedup grows with the input size and that the smaller bbSCITE variant achieves a higher throughput for the inputs it can process than the bigger variant, despite slightly lower clock frequency. This means that if a user knows that their input will not exceed a certain synthesizable size, it can be beneficial to only build bbSCITE with this bit-vector length.

4.3 Performance model and pipeline latency

The throughput of bbSCITE can be modeled by the number of iterations in its pipelined loops, ideally consuming one cycle per iteration. The slowest of these loops transfers the current state of a chain between IO and worker kernel. Each state is therefore transferred in packets, row by row. With n denoting the number of nodes in the mutation tree (i.e. the number of utilized bit-vector bits and rows in an ancestor matrix), then the first n packets contain a tuple with a row of the ancestor matrix and a row of the descendant matrix, followed by one packet with meta information. In total, there are $n+1$ packets per state, which takes $n+1$ cycles to be transferred. The other pipelined loops, i.e. the likelihood calculation and the mutation tree updates, should be slightly faster at n cycles. Thus, the throughput of the pipe in steps per second is upper bounded by $f/(n+1)$ in steps per second, where f is the clock frequency of the final design.

We compare the measured throughput of bbSCITE with this model in Tab. 3. The model matches fairly well for inputs that fully utilize the available bit-vectors, but the model accuracy drops off for smaller inputs. We suspect that this might be due to the overall latency of the worker kernel: The IO kernel currently assumes a fixed latency of the worker kernel, measured in chain states, which does not take the input size into account. However, the worker's latency in processed chain states might be bigger for smaller input sizes. This would mean that the IO kernel needs to stall until the results are sent back from the worker, which would explain the lowered throughput. This hypothesis could be investigated with further experiments. The presented designs are based on a coarse empirical design space exploration, which yielded 24 independent chain states to hide the worker latency. The good correspondence of measurements and model for full inputs confirms that this choice is reasonable for this case, but may need refinement for other input sizes.

[1]https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-0/var-prec-fp-sup.html

Kernel/Component	Look-up tables		Flip-Flops		RAM blocks		MLABs		DSPs	
	64-bit	**96-bit**	**64-bit**	**96-bit**	**64-bit**	**96-bit**	**64-bit**	**96-bit**	**64-bit**	**96-bit**
Static Partition	24.4%	24.4%	24.4%	24.4%	22.4%	22.4%	0.0%	0.0%	18.2%	18.2%
Interconnect	1.2%	1.2%	0.7%	0.7%	1.0%	0.9%	0.0%	0.0%	0.0%	0.0%
IO Kernel	0.7%	0.6%	0.6%	0.6%	0.8%	0.6%	<0.1%	<0.1%	<0.1%	<0.1%
Worker Kernel	16.4%	27.7%	10.5%	13.6%	6.5%	10.8%	1.0%	0.6%	9.8%	12.6%
(thereof "Execute Move")	*(9.2%)*	*(13.6%)*	*(5.4%)*	*(6.3%)*	*(3.0%)*	*(2.9%)*	*(0.6%)*	*(0.3%)*	*(8.0%)*	*(10.8%)*
Total	**42.7%**	**53.9%**	**36.2%**	**39.4%**	**30.7%**	**34.9%**	**1.0%**	**0.6%**	**28.0%**	**30.8%**

Table 1: Resource usage of the 64-bit and 96-bit design instances. The 64-bit instance is clocked at 307.50 MHz, and the 96-bit instance is clocked at 316.67 MHz. Percentages are relative to available resources on the Intel Stratix 10 GX 2800 FPGA (1866240 look-up tables, 3732480 flip-flops, 11721 RAMs, ~133900 MLABs, 5760 DSPs). The static partition provides the interface to the host and is fixed by the card vendor.

Input Size	Throughput [ksteps/s]			Speedup	
Cells × Genes	CPU	FPGA design		FPGA/CPU	
		96-bit	64-bit	96-bit	64-bit
32×31	229.87	4891.25	7486.57	21×	33×
64×63	70.65	4123.53	4349.70	58×	62×
95×94	36.55	2938.25	n/a	80×	n/a

Table 2: Mean throughput of SCITE (single thread on AMD EPYC Milan 7763 CPU) and two variants of bbSCITE (Stratix 10 GX 2800 FPGA) for different input sizes. 64-bit bbSCITE is not able to process mutation matrices with more than 63 genes since the bit vector width limits the number of genes it can process.

Instance	Input Size	Throughput [ksteps/s]		Acc.
	Cells × Genes	Measured	Modelled	
96-bit	32×31	4891.25	9596.06	51.0%
	64×63	4123.53	4871.85	84.6%
	95×94	2938.25	3298.65	89.1%
64-bit	32×31	7486.57	9318.18	80.3%
	64×63	4349.70	4730.77	91.9%

Table 3: Comparison of measured and modeled throughput for different design instances and input sizes, including the model accuracy.

5 ANALYSIS WITH IDEALIZED CPU REFERENCE

In Subsection 4.2, we have shown that bbSCITE achieves up to 80× the throughput of the CPU reference implementation by Jahn et al. [3]. This is a good result, but strongly influenced by limited performance engineering done for the reference. In particular, it does not make use of multi-threading to parallelize the work and also uses many memory allocation calls within loop kernels. In this section, we try to assess how bbSCITE could compare to a highly optimized CPU version, without undergoing the CPU optimization effort for the full application. Therefore, we focus on the likelihood calculation as a synthetic proxy to upper-bound the optimized CPU throughput.

Our synthetic benchmark runs the log-likelihood computation (Alg. 2) on mutation matrices with 128 cells and 127 genes: It first loads the mutation data matrix and prepares the initial Markov chain states like a functional implementation of SCITE. However, it only repeatedly computes the likelihood score of these initial states and never modifies them. We chose the log-likelihood function as our kernel since it is the part of the algorithm with the fastest-growing runtime and also consumes a significant resource fraction of the FPGA designs. In contrast to the largest evaluated FPGA design with 96 bits, we chose 128 cells and 127 genes as the input size, since power-of-two dimensions often enable more efficient CPU execution, in particular when vector operations are used. The benchmark is also implemented in C++ using SYCL and Intel OneAPI and shares much of the non-performance-critical code with bbSCITE.

We ran the benchmark on individual nodes of the two supercomputers Noctua 1 and Noctua 2, hosted at the Paderborn Center for Parallel Computing. Noctua 1 nodes, installed in 2018, contain 2 Intel Xeon Gold 6148 CPUs that are manufactured with the same 14 nm technology as the Intel Stratix 10 GX 2800 FPGA we used for bbSCITE. Noctua 2 nodes contain 2 AMD EPYC Milan 7763 CPUs, representing a high-end server CPU platform in 2022. Most of the operations in the loop body can be vectorized except for the popcount instruction. This means that in a vectorized implementation of the log-likelihood function, the vectors are completely unpacked for the popcount operations and are then repacked afterwards. With the 512-bit vector registers of the Noctua 1 nodes, the total number of executed instructions is reduced in comparison to a scalar implementation, but with the 256-bit vector registers of the Noctua 2 nodes, this is not the case. We have therefore decided to use the respectively faster vectorized implementation for Noctua 1 and the scalar implementation for Noctua 2. Apart from this, we have used the IntelLLVM compiler version 2022.2 for Noctua 2 and version 2022.2.1 for Noctua 1. The compiler arguments were

```
-O3 -DNDEBUG -qactypes -std=gnu++20
-fsycl-targets=spir64_x86_64-unknown-unknown
```

for all operations, and additionally

```
-Xsdevice=cpu -Xsmarch=avx2
```

Hardware (Design)	Input Size [cells × genes]	Throughput [counted Tbit/s]	Power Draw [W]	Energy Consumption [pJ/counted bit]
2x AMD EPYC Milan 7763	128 × 127	18.88	556.05	29.45
1x Stratix 10 GX 2800 (96-bit)	95 × 94	10.08	75.17	7.46
1x Stratix 10 GX 2800 (64-bit)	64 × 63	4.56	72.42	15.88
2x Intel Xeon Gold 6148	128 × 127	1.95	**300.00*	**153.85*

Table 4: Comparison of synthetic log-likelihood calculation on CPU vs. full bbSCITE on FPGA. Power and energy numbers based on measured power consumption, except for **marked* numbers based on nominal TDP.

for linking on Noctua 2, and

```
-Xsdevice=cpu -Xsmarch=avx512
```

for linking on Noctua 1.

For the 128 × 127 inputs, one entire Noctua 2 node achieves a throughput of 2251.02 ksteps/s and one entire Noctua 1 node achieves 233.05 ksteps/s. Both systems are utilized very well: Noctua 1 achieves an L1D miss-rate of 0.03% and an IPC of roughly 1.87, while Noctua 2 even achieves an L1D miss-rate of 0.005% and an IPC of roughly 5.1. We assume that there may be some more optimization possible by reducing the number of instructions, but we should also stress that this performance does not even cover the entire Markov chain step

Since the throughput in ksteps/s varies on all architectures with different input sizes, as seen in Section 4.2, we also need to normalize between the synthetic CPU benchmark and the full bbSCITE measurements. To this end, we introduce the metric of counted bits per second. Every chain step requires preparing and counting the 1s in $4 \times n^3$ bits where n is the number of cells and the number of bits in a bit-vector. By multiplying the number of counted bits per chain step with the measured throughput in steps per second, we obtain the performance figures in Tab. 4. When comparing individual chips, we see that 96-bit bbSCITE performance is en-par with a single AMD EPYC Milan CPU (note that Tab. 4 presents numbers for the entire node) and outperforms a single Intel Xeon Gold CPU by an order of magnitude. Looking at power measurements and energy efficiency, we see that the two EPYC Milan CPU per node consume around 7.5× more power than a single FPGA board. Normalized to energy consumption per counted bit, with 7.5 pJ/bit the FPGA architecture is around 4× more energy efficient than the EPYC Milan CPU at 29.5 pJ/bit.

6 RELATED WORK

Typical FPGA accelerators for bioinformatics workloads [2, 5, 6] rely solely on highly data-parallel datapaths with customized bitwidth. Similar to our work, Guo et al. [1] additionally consider an iterative step for a form of chain expansion around the data-parallel core. With this, they also encounter a latency challenge on the feedback of this step. Orthogonal to our solution of overlapping this latency with steps from other chains, they focus on minimizing the latency of the individual step by reordering the computation order.

On the algorithmic side, Kuipers et al. [4] have extended the search space for possible mutation trees by softening the infinite-site assumption. This allows to reconstruct some mutation histories more accurately and could be considered in possible extensions of bbSCITE.

7 CONCLUSION AND FUTURE WORK

In this work, we have introduced bbSCITE, an FPGA design and implementation of the SCITE algorithm that outperforms the original CPU reference by up to 80×. For inputs matching the design parameters, a performance model illustrates the good occupancy of the FPGA design. We have also approximated the maximum performance that an optimized CPU implementation can achieve for one of the core loops of bbSCITE. Even here, the FPGA performance is en-par with current high-performance CPUs at much lower power consumption, while outperforming CPUs of the same manufacturing technology by an order of magnitude.

We see several directions for future work. A main limitation of the current design are the maximal input and mutation matrix sizes fixed at compile-time. A blocking approach could open up the design for arbitrarily large inputs, possibly at the cost of decreased performance. On the other hand, since the presented designs still leave a signification fraction of FPGA resources unused, there could be headroom for further performance improvements, for example via replications of the worker kernel.

REFERENCES

[1] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 127–135. https://doi.org/10.1109/FCCM.2019.00027

[2] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. 2008. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 1, 2, Article 9 (June 2008), 44 pages. https://doi.org/10.1145/1371579.1371581

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

[4] Jack Kuipers, Katharina Jahn, Benjamin J Raphael, and Niko Beerenwinkel. 2017. Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors. *Genome research* 27, 11 (2017), 1885–1894. https://doi.org/10.1101/gr.220707.117

[5] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong. 2003. A Smith-Waterman Systolic Cell. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL) (LNCS, 2778)*. Springer, Berlin / Heidelberg, 375–384. https://doi.org/10.1007/b12007

[6] Peiheng Zhang, Guangming Tan, and Guang R. Gao. 2007. Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform. In *Proc. Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)* (Reno, Nevada). Association for Computing Machinery (ACM), New York, NY, USA, 39–48. https://doi.org/10.1145/1328554.1328565