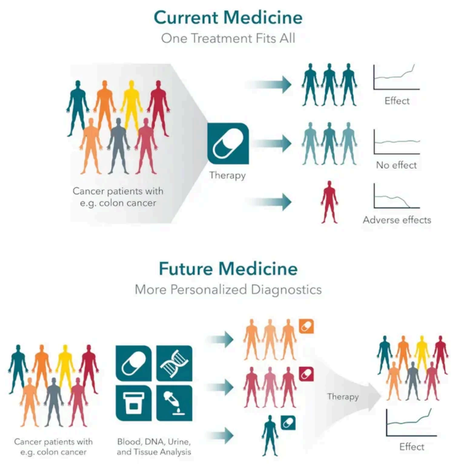
HARDWARE ACCELERATION FOR GENOMICS

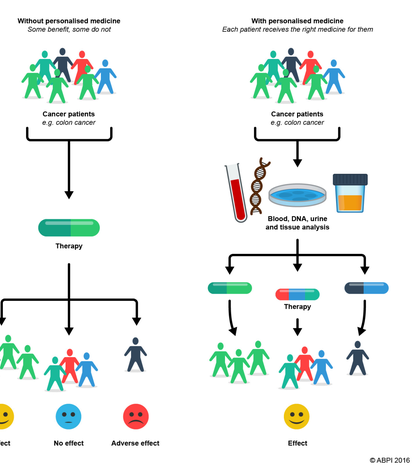
Department of Electrical Engineering, Indian Institute of Technology Gandhinagar

Anuja Chaudhari
Sia Jariwala
Ginisha Garg
Seemanshi Mall

1. Motivation

Genomics decodes the entire DNA sequence of an organism to uncover insights into genes, diseases, evolution, and personalized medicine.





Why Hardware Acceleration?

- Massively Parallel Processing: FPGAs handle multiple tasks simultaneously for ultra-fast analysis.
- Energy-Efficient Computing: Lowers power usage — ideal for edge and embedded platforms.
- Real-Time Alignment: Enables immediate processing of reads as they are sequenced.
- Enhanced Privacy: Keeps genomic data on-device, reducing risk of external breaches.

Challenges with Traditional Software-Based Genomic Analysis

- Slow Processing: Software alone struggles with massive, ever-growing genomic datasets.
- High Computational Cost: Demands intensive CPU time and energy.
- Data Privacy Concerns: Sensitive DNA data often sent to external cloud servers.
- Scalability Limits: Poor adaptability as sequencing throughput scales up.

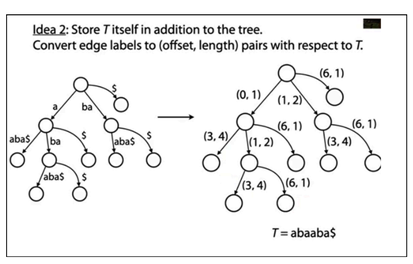
Practical Applications of Hardware-Accelerated Genomics

- Clinical Diagnostics: Rapid identification of genetic disorders, cancer mutations, and pathogen genomes for real-time medical decisions.
- Personalized Medicine: Tailoring treatment plans based on an individual's genomic profile — enabling more effective, targeted therapies.
- Pandemic Response: Fast sequencing and tracking of viral genomes (e.g., COVID-19 variants) to support vaccine and drug development.
- Ancestry & Evolutionary Studies: Large-scale comparison of genomes across populations/species to trace human ancestry and evolutionary patterns.
- Forensic Genomics: Rapid DNA matching for criminal investigations, missing persons identification, and disaster victim recovery.

2. Exact Matching

Build Suffix Tree from Text
Input: Text T of length n
Output: Root node of the Suffix Tree

```
1 Function buildSuffixTree(T):
2  root ← newNode()
3  for i ← 0 to length(T) - 1 do
4   current ← root
5   j ← i
6   while j < length(T) do
7    c ← T[j]
8    if c not in current.children then
9     child ← newNode()
10     current.children[c] ← (j, length(T) - j, child)
11     child.suffix_index ← i
12     break
13    else
14     start, length, child ← current.children[c]
15     k ← 0
16     while k < length and j + k < length(T)
and T[start + k] = T[j + k] do
17      k ← k + 1
18     if k = length then
19      current ← child
20      j ← j + k
21     else
22      split ← newNode()
23      current.children[c] ← (start, k, split)
24      split.children[T[start + k]] ← (start + k,
length - k, child)
25      newLeaf ← newNode()
26      split.children[T[j + k]] ← (j + k, length(T) -
(j + k), newLeaf)
27      newLeaf.suffix_index ← i
28      break
29  return root
```



Compute C Table
Input: BWT
Output: C Table

```
1 Function computeCTable(BWT):
2  sortedBWT ← sort(BWT)
3  uniqueChars ← sorted(set(BWT))
4  for char in uniqueChars do
5   C[char] ← first index of char in
sortedBWT
6  return C
```

Compute OCC Table
Input: BWT
Output: OCC Table, Character Index Map

```
1 Function computeOCCTable(BWT):
2  uniqueChars ← sorted(set(BWT))
3  occTable ← 2D array of zeros
[len(uniqueChars)][len(BWT) + 1]
4  charToIndex ← map each char to index
5  for i ← 1 to len(BWT) do
6   occTable[:, i] ← occTable[:, i - 1]
7   char ← BWT[i - 1]
8   occTable[charToIndex[char], i] ←
occTable[charToIndex[char], i] + 1
9  return occTable, charToIndex
```

Generate Suffix Array from Suffix Tree
Input: Root of Suffix Tree
Output: Suffix Array SA

```
1 Function getSuffixArray(root):
2   result ← emptyList()
3   Function DFS(node):
4     if node.suffix_index ≠ -1 then
5       result.append(node.suffix_index)
6     for key in sorted(node.children) do
7       _, _, child ← node.children[key]
8       DFS(child)
9   DFS(root)
10   return result
```

Compute BWT from Suffix Array
Input: Text T, Suffix Array SA
Result: Burrows-Wheeler Transform BWT

```
1 Function computeBWT(T, SA):
2  T ← T + "$"
3  BWT ← ""
4  for i in SA do
5   if i = 0 then
6    BWT ← BWT + "$"
7   else
8    BWT ← BWT + T[i - 1]
9  return BWT
```

Backward Search using FM-Index
Input: Pattern P, C Table, OCC Table, charToIndex, BWT
Output: List of match indices in Suffix Array

```
1 Function backwardSearch
(P, C, OCC, charToIndex, BWT):
2  s ← 1
3  e ← len(BWT)
4  for char in reverse(P) do
5   if char not in charToIndex then
6    return []
7   idx ← charToIndex[char]
8   s ← C[char] + OCC[idx][s - 1] + 1
9   e ← C[char] + OCC[idx][e]
10   if s > e then
11    return []
12  return range(s - 1, e)
```

Reconstruct Read from Seeds
Input: List of seeds, reference genome, FM-index data
Output: Reconstructed sequence

```
1 Function reconstructFromSeeds(seeds,
refGenome, SA, C, OCC, charToIndex, BWT):
2  reconstruction ← list of '-' of
length(refGenome)
3  seedCount ← count each seed in seeds
4  for seed in seedCount:
5   matches ← backwardSearch(seed, C,
OCC, charToIndex, BWT)
6   positions ← [SA[i] for i in matches]
7   used ← 0
8   for pos in positions do
9    if used ≥ seedCount[seed] then
10     break
11    if pos + len(seed) ≤ length(refGenome)
then
12     reconstruction[pos:pos+len(seed)] ←
list(seed)
13    used ← used + 1
14  return join(reconstruction)
```

Applications of exact matching and fast backward search of query reads

Mutation Detection: Allows for quick identification of genetic mutations, enabling fast diagnostics and personalized treatments.

Some cancers or other diseases have specific mutations .Quick detection of these mutations can help doctors choose targeted therapies and avoid ineffective treatments.

Accelerating Gene Expression Analysis: Quickly Mapping RNA Reads for Cancer Research and Identifying Potential Biomarkers

CRISPR Off-Target Detection: Ensures accurate genome editing by minimizing off-target effects, enhancing the safety of gene therapies. It is widely used in crop improvement and in developing genetically modified animals for research.

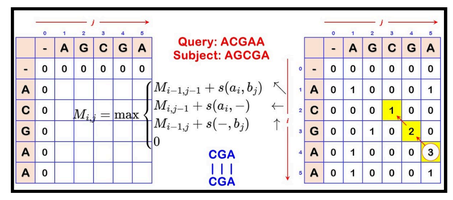
3. Approximate Matching

Base-4 Rolling Hash Functions for DNA k-mer Indexing
Inputs: K-mer(A DNA substring of length k), base (4 for DNA nucleotides)
Output: Hash value

```
1: Define mapping ← {A: 0, T: 1, C: 2, G: 3}
2: Function BASE4_HASH(kmer, base):
3:   h ← 0
4:   for each char in kmer do
5:     h ← h * base + mapping[char]
6:   return h
7: Function ROLLING_HASH(prev_hash, out_char,
in_char, k, base):
8:   out_val ← mapping[out_char]
9:   in_val ← mapping[in_char]
10:  prev_hash ← prev_hash - out_val * base^(k - 1)
11:  prev_hash ← prev_hash * base + in_val
12:  return prev_hash
```

Local Alignment, Base Aggregation, and Consensus Generation
Input: Reference Genome, reads and candidate position for each read
Output: ref_alignments: A map of reference positions to aligned read bases.
consensus_sequence: A sequence of the most frequent base at each position in ref_alignments.

```
Initialize ref_alignments = empty map: reference
position → list of aligned read bases
For each read and its candidate positions:
 Initialize best_score = -∞
 For each candidate position:
 Extract reference segment starting at that position
 Align read with segment using SMITH_WATERMAN
 Calculate alignment score
 7. If score > best_score:
 8. Update best_score
 9. Store best_alignment and best_position
 10. If best_position is found:
 11. Print aligned read and reference segment
 12. For each aligned base pair in best_alignment:
 13. If reference base is not a gap:
 14. Append read base to ref_alignments at
corresponding reference position
# Generate consensus sequence
Initialize consensus_sequence as empty string
For each reference position in sorted(ref_alignments):
 If list of aligned bases is not empty:
 Find the most frequent base at that position
 Append it to consensus_sequence
 Else:
 Append 'N' to consensus_sequence
```



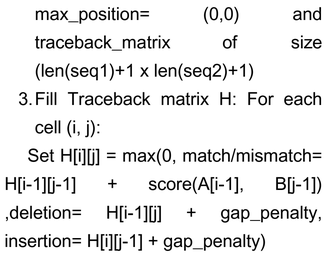
Global Positioning Algorithm
Input: Reference genome, sequenced reads
Output: Probable matching regions in the reference for each read

```
1. Function
   FIND_MATCHING_REGIONS(reference,
   reads, k):
2.  Initialize seed_table = empty map
3.        hash = BASE4_HASH(first k-mer of
   reference)
4.  Store position 0 under hash

5.  For i from 1 to len(reference) - k:
6.  hash = ROLLING_HASH(hash, reference[i-
1], reference[i+k-1], k)
7.  Store position i under hash

8.  For each read:
9.      hash = BASE4_HASH(first k-mer)
10          Get all positions from seed_table
matching the hash
11.     Return map of read → list of candidate
positions
```

Final Reconstruction
Input: reference_genome , ref_alignments
Output: Reconstructed Genome

```
Initialize reconstructed_reference as empty string

1. For each position i from 0 to length of
   reference_genome - 1:
2.  If ref_alignments[i] is not empty:
3.          Count occurrences of each base in
   ref_alignments[i]
4.      Select the base with the highest frequency
   (most common base)
5.                  Append that base to
   reconstructed_reference
6.  Else:
7.          Append '-' to reconstructed_reference to
   indicate missing coveragev
```

Genetic mutations



Smith Waterman Algorithm
Input: Two sequences to be aligned locally.
Output: The optimal local alignment between the two sequences

- Define match reward =+2, mismatch penalty = -1, gap penalty = -1
- Initialize scoring matrix (4x4) for base alignment

```
Function GET_SCORE(base1, base2):
                Return value from
scoring_matrix[base1][base2]

Function SMITH_WATERMAN(seq1,
seq2):
1. Initialize 2 rows: previous_row,
   current_row (size = len(seq2) + 1)
2. Initialize max_score=0 and
   max_position= (0,0) and
   traceback_matrix of size
   (len(seq1)+1 x len(seq2)+1)
3. Fill Traceback matrix H: For each
   cell (i, j):
  Set H[i][j] = max(0, match/mismatch=
H[i-1][j-1] + score(A[i-1], B[j-1])
,deletion= H[i-1][j] + gap_penalty,
insertion= H[i][j-1] + gap_penalty)

  4. Start traceback from the cell with
the maximum score, move:
  Diagonally if the current cell was
derived from a match/mismatch, Up if
from a deletion, Left if from an insertion
  6. Stop when a cell with value 0 is
reached and construct the aligned
sequences during the traceback
```

Applications of Approximate Matching:

Error-Tolerant Read Mapping: Enables accurate alignment of DNA reads that contain sequencing errors or biological variations, improving the sensitivity and reliability of genome analysis tools.
Detection of Genetic Variants: Facilitates identification of SNPs, insertions, deletions, and other structural variants, even when reads deviate slightly from the reference genome — critical for population genetics and disease research.
Robust Genome Assembly: Helps reconstruct genomes from fragmented or noisy reads by allowing flexible overlaps, ensuring more complete and accurate assemblies in de novo sequencing.

4. Flowchart