Domain-Specific Many-Core

CHIPS
PES UNIVERSITY

Centre for
Heterogeneous and
Intelligent
Processing
Systems

PES University | Electronic City Campus

CPU

GPU

FPGA

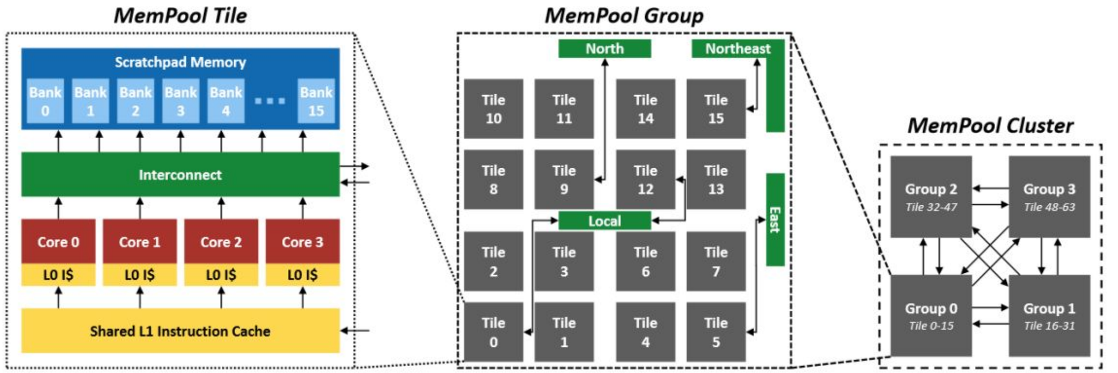
ASIC

www.chips.pes.edu

- Abhiram Gopal Dasika [PES2UG21EC003]
- Jitesh Kumar Nayak [PES2UG21EC057]
- Neha C Waghmore [PES2UG21EC092]

Review

Literature: Mempool - A RISC-V Many-core cluster[1]



Mempool Tile:

- 4 cores
- 16 memory banks
- Single-cycle latency

Mempool Group:

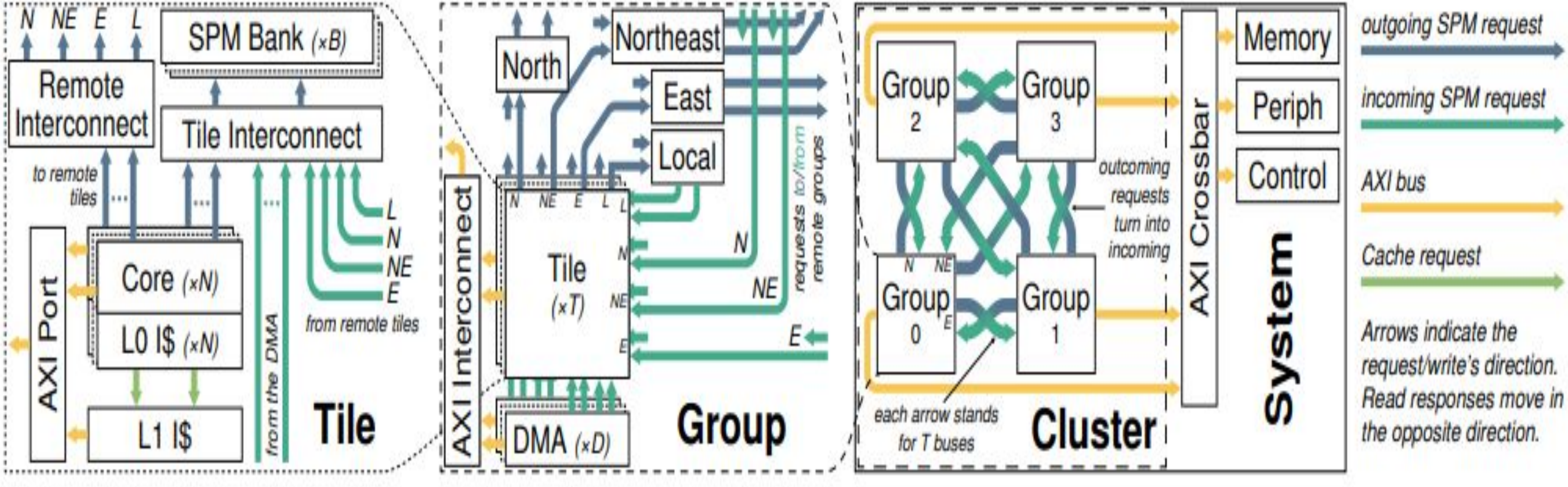
- 64 cores
- 256 memory banks
- 3 cycles latency

Mempool Cluster:

- 256 cores
- 1024 memory banks (1MiB)
- 5 cycles latency

- Why MemPool?
- Is an easily configurable, hugely scalable, multiple core cluster, having shared L1 caches

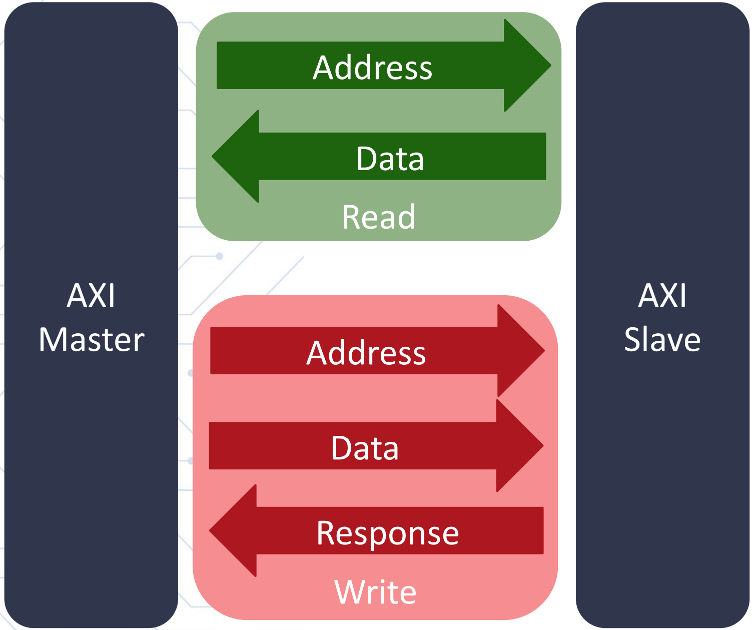
Mempool architecture[1]



Arrows indicate the request/write's direction. Read responses move in the opposite direction.

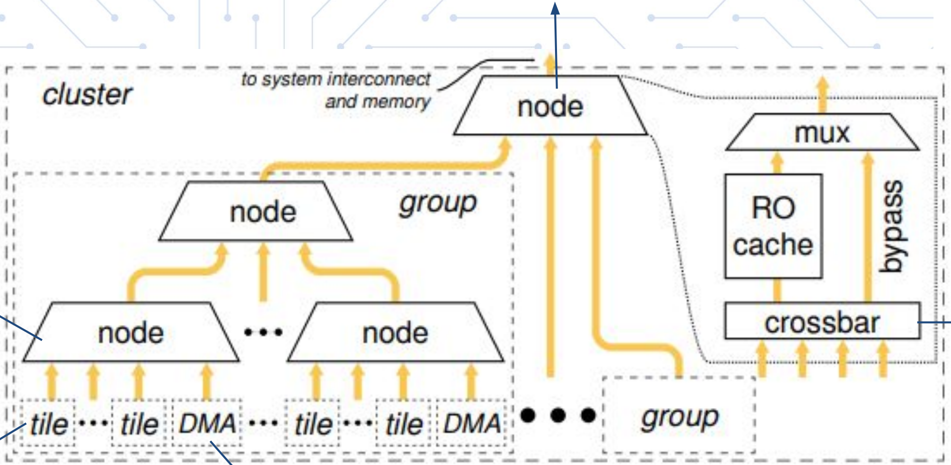
AXI-interconnect

- Defined in AMBA standard by ARM, AXI is an open interface protocol which stands for Advanced eXtensible Interface
- AXI can be AXI4 (memory-mapping), AXI4-Lite (low-throughput memory-mapping) or AXI4-Stream (data streaming)
- AXI protocol defines 5 channels, where 2 are used for Read transactions (`read address`, `read data`) and 3 are used for Write transactions (`write address`, `write data`, `write response`)



AXI-interconnect hierarchy[1]

Each tile has an AXI master port shared between all cores and the inst. cache refill giving them access to higher level main memory, peripherals and control registers



AXI interconnect

The local crossbar is an interconnect block used to communicate between cores and the scratchpad memories in the tile

Each tile has an AXI port that goes to the interconnect

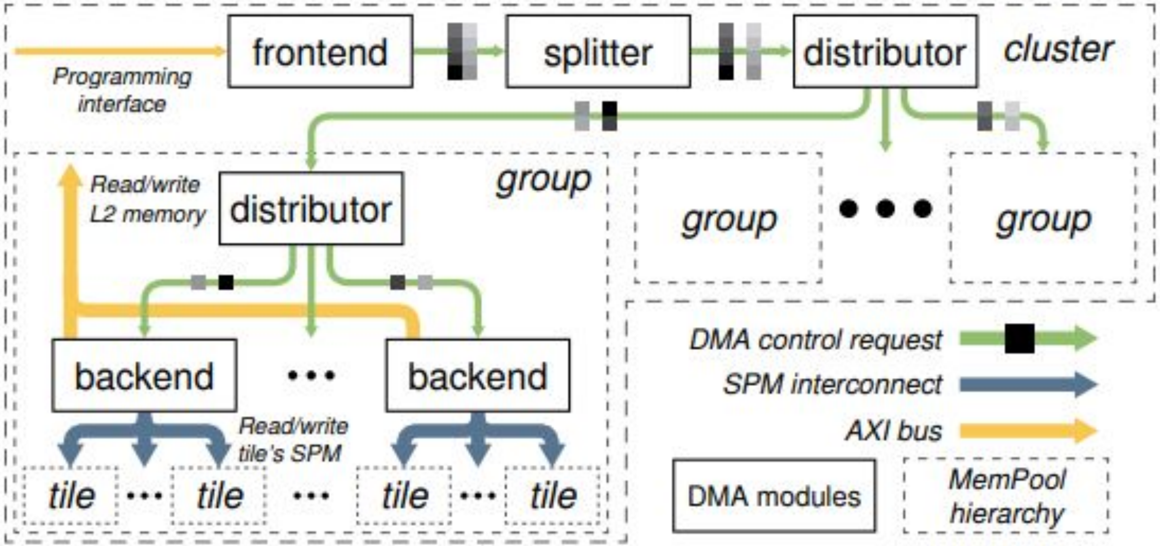
DMA Engine has access to AXI interconnect and the local crossbar

DMA Engine

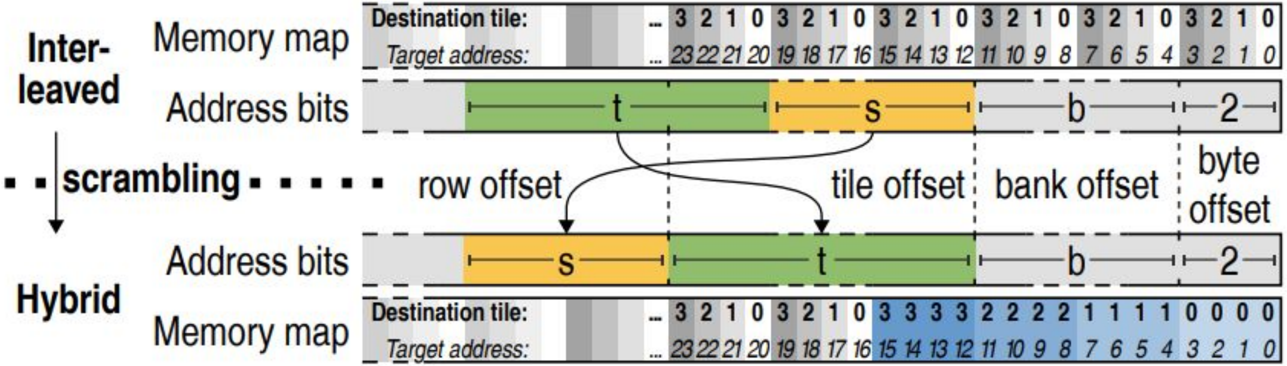
- DMA (Direct Memory Access) engine.
- DMA engine in MemPool is a subsystem that enables data transfer between devices and memory without involving the CPU.
- Owns some set of registers and controls, that allow it to access memory directly, bypassing the CPU's involvement for data movement.
- This can significantly improve system performance by offloading data transfer tasks from the CPU to the DMA engine, which can handle them more efficiently.
- DMA engine might be responsible for managing memory allocation and deallocation for devices that need to transfer data to and from memory(Data Modules).
- It can help in optimizing memory usage and reducing latency for data transfers.

Centre for
Heterogeneous and
Intelligent
Processing
Systems

DMA engine[1]



Hybrid addressing[1]



Inter-leaved memory map:

Destination tile:	...	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
Target address:	...	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

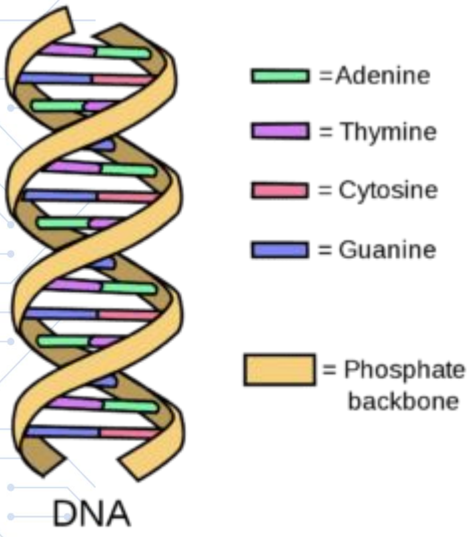
Hybrid memory map:

Destination tile:	...	3	2	1	0	3	2	1	0	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
Target address:	...	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

- When a core has to access contiguous locations it has to shift banks increasing latency
- Sequential regions are created in every bank

Background on DNA

- A genome is an organism's complete set of genetic instructions that is stored in the DNA
- The building blocks of DNA are called nucleotides which are molecules of sugar, a nitrogenous base and a phosphate group.
- Four Main Nitrogenous Bases-
    - Adenine (A)
    - Guanine (G)
    - Cytosine (C)
    - Thymine (T)
- These 4 Bases arranged in double Helix Structure makes one strand of DNA.



Centre for Heterogeneous and Intelligent Processing Systems

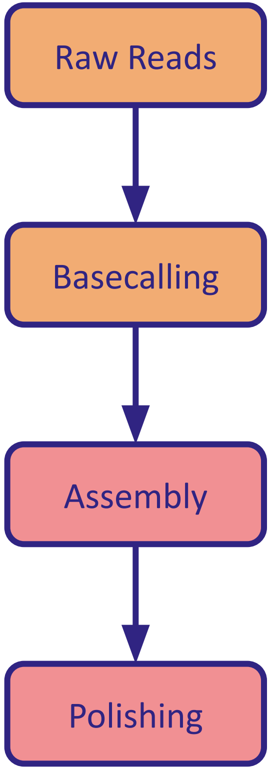
Genomic Pipeline[1]

Raw Reads

- Raw Reads are the raw electrical signals that were produced from the sequencing machine.
- These are usually stored in `.fast5` file format.
- The .fast5 files that were used in this project were from the organism - *Klebsiella pneumoniae*.

Basecalling

- Basecalling is a process which translates the raw electrical signals from the ONT Sequencer into nucleotide bases using RNN's.
- Guppy from Oxford Nanopore Technologies was used for basecalling
- We obtain a `.fastq` file once basecalling is completed.



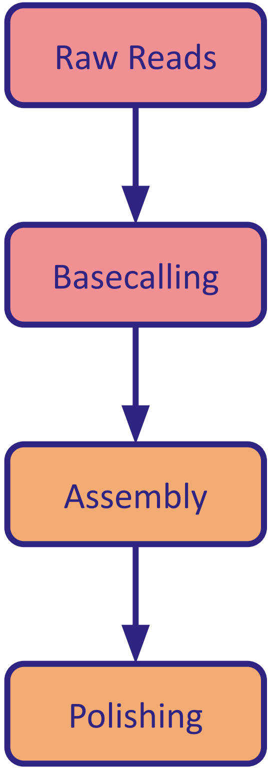
Genomic Pipeline[1]

Assembly

- During sequencing, the DNA is chopped up into small fragments called reads.
- Assembler takes these reads and reassembles them to reconstruct the original sequence.
- There are 2 ways to perform assembly
  - **Reference Assembly:** Assembles reads by mapping the reads against a reference sequence/genome.
  - **De Novo Assembly:** Assembles reads without using any kind of template or reference sequence/genome.

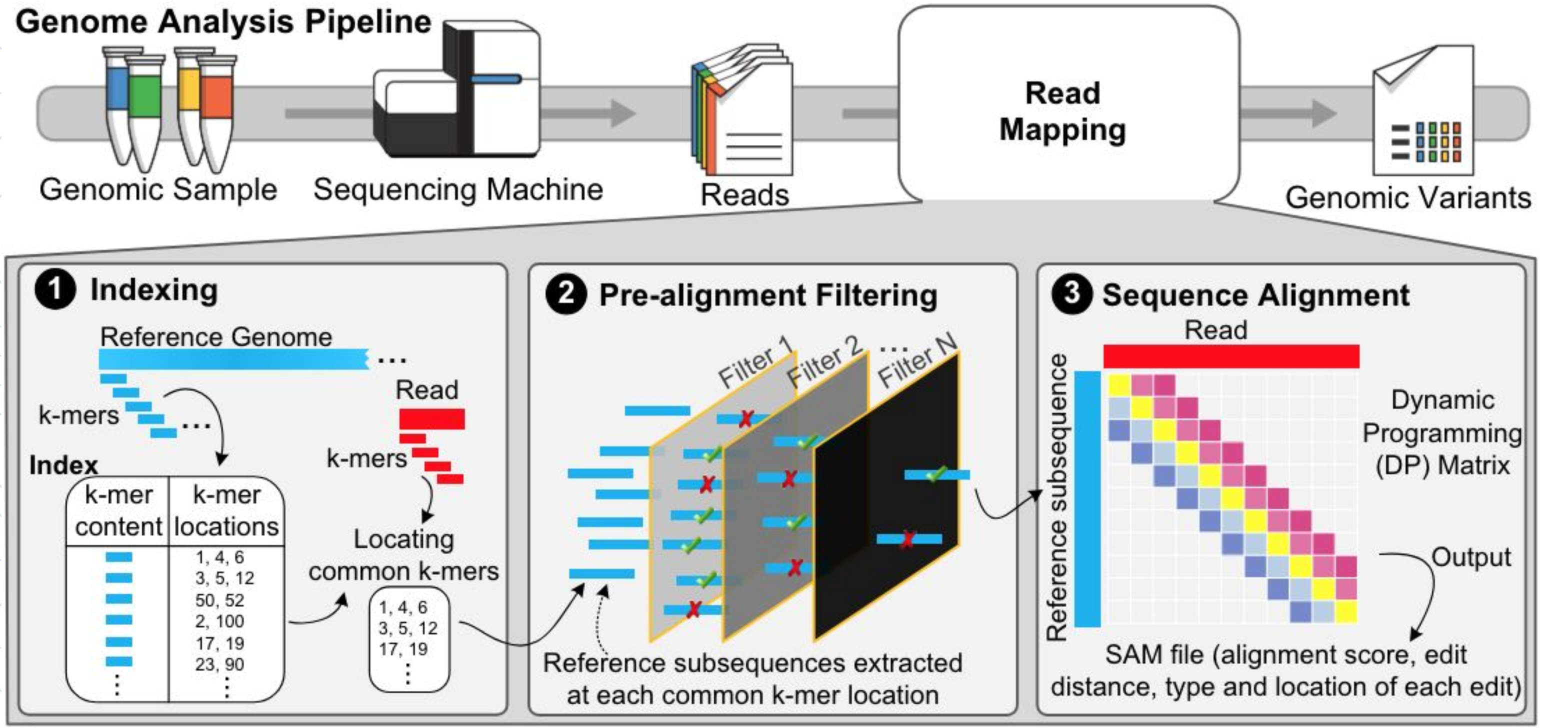
Polishing

- Removing/correcting the errors that are formed in the assembled sequence.
- The output of polishing is a `.fasta` file

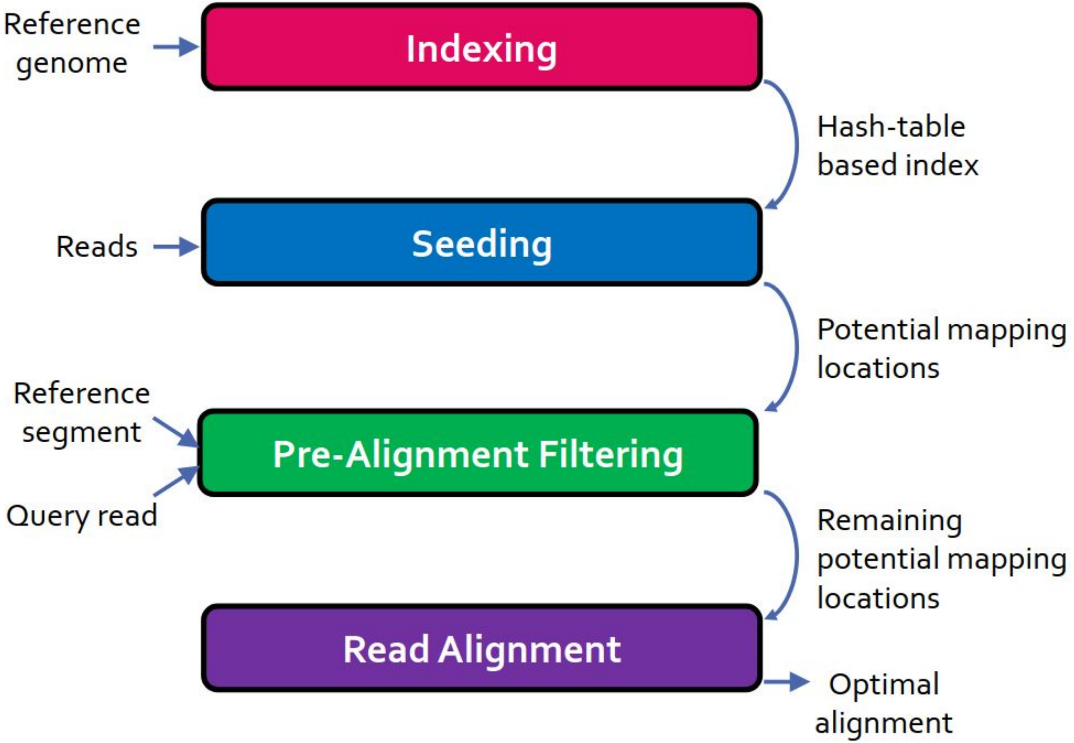


What is genome sequencing?[1]

- Figuring out the exact order of the base pairs.

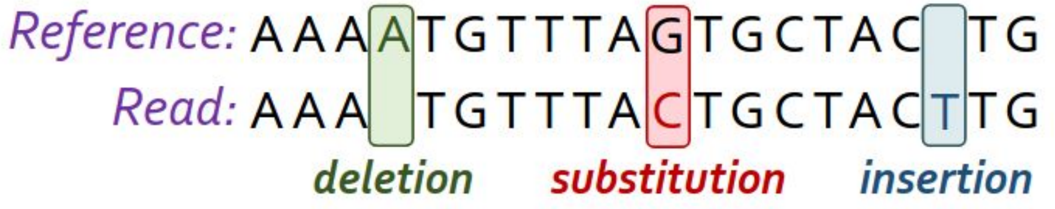


Read mapping

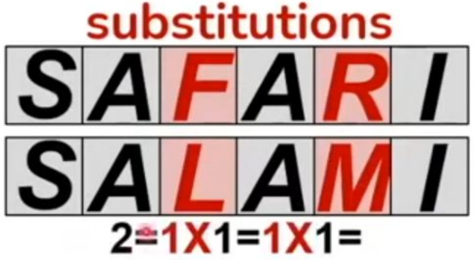


Pre-requisites

1. Approximate String Matching involves matching the query string with the reference string allowing few edits these edits can be either deletion substitution and insertion



2. Cigar string tells us what edits to make and where do we have to make them



GenASM

- GenASM is an approximate string matching (ASM) acceleration framework for genome sequence analysis
- GenASM-DC (Data computation) is responsible for searching sub-patterns within sub-texts
- GenASM-TB handles the traceback operation
- GenASM enhances the Bitap bitwise operations that are used in approximate string matching

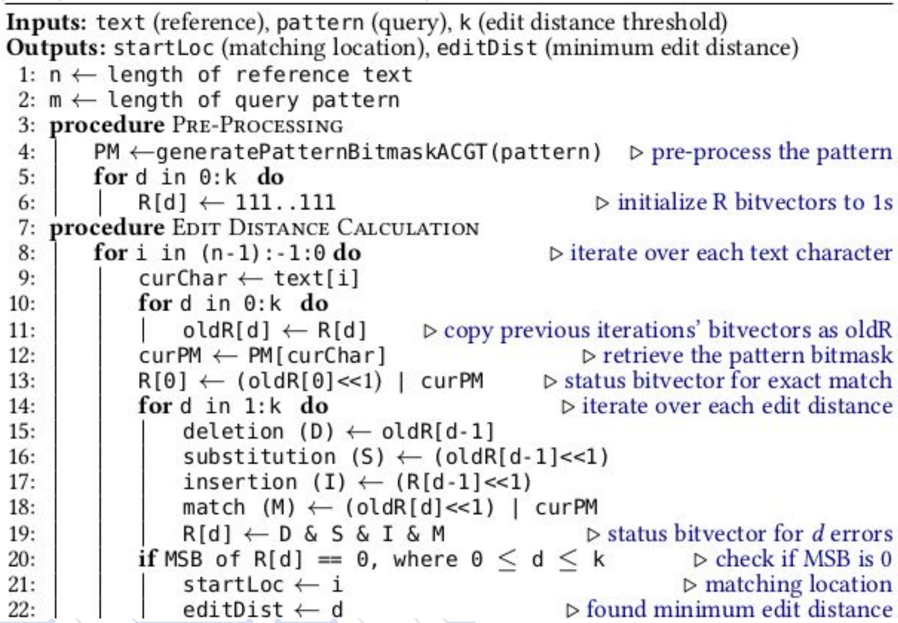
GenASM DC

Inputs:

1. Query string
2. Reference string

Outputs:

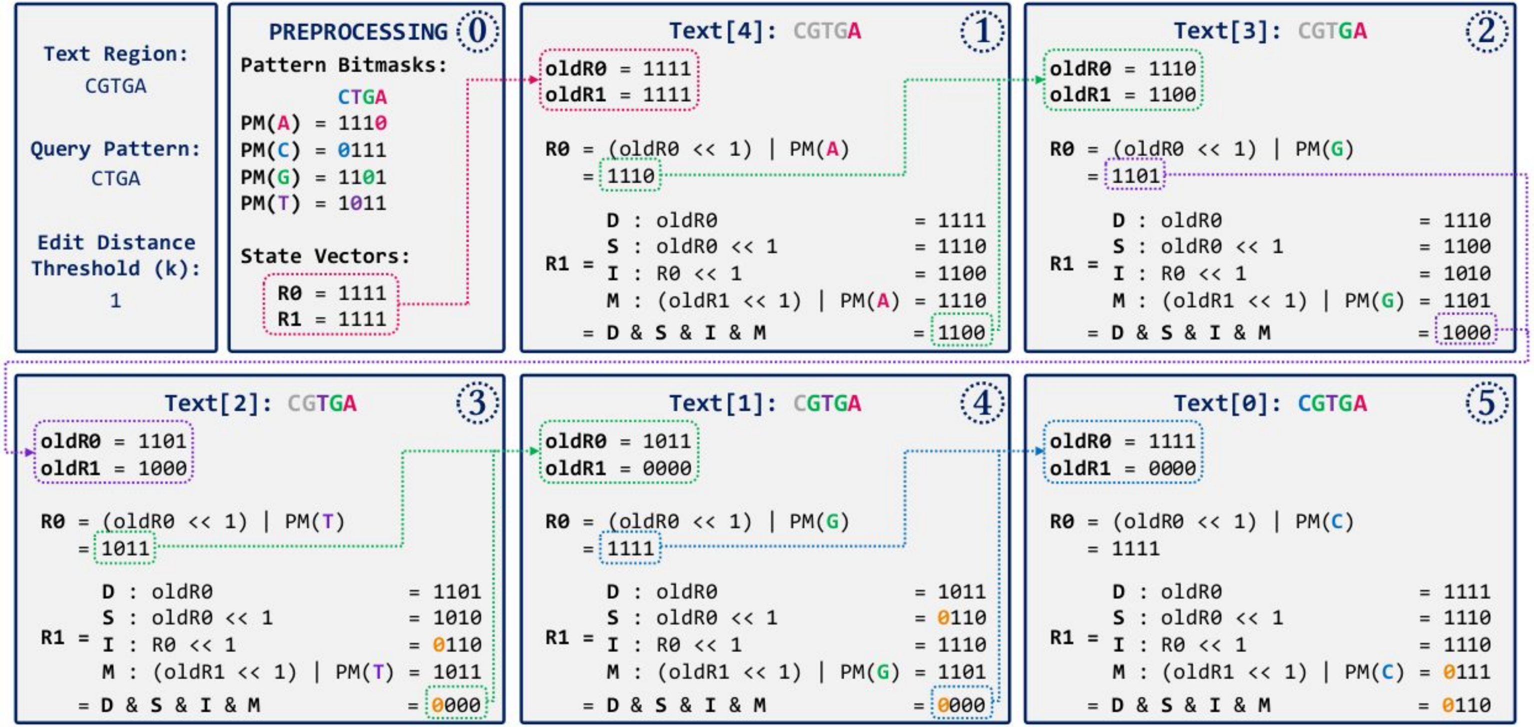
1. Minimum edits to match with reference string
2. Location of the match
3. Traceback matrix



Inputs: text (reference), pattern (query), k (edit distance threshold)
Outputs: startLoc (matching location), editDist (minimum edit distance)

```
n ← length of reference text
m ← length of query pattern
procedure PRE-PROCESSING
    PM ←generatePatternBitmaskACGT(pattern)          ▷ pre-process the pattern
    for d in 0:k do
        R[d] ← 111..111                              ▷ initialize R bitvectors to 1s
procedure EDIT DISTANCE CALCULATION
    for i in (n-1):-1:0 do                           ▷ iterate over each text character
        curChar ← text[i]
        for d in 0:k do
            oldR[d] ← R[d]          ▷ copy previous iterations' bitvectors as oldR
        curPM ← PM[curChar]                          ▷ retrieve the pattern bitmask
        R[0] ← (oldR[0]<<1) | curPM                  ▷ status bitvector for exact match
        for d in 1:k do                              ▷ iterate over each edit distance
            deletion (D) ← oldR[d-1]
            substitution (S) ← (oldR[d-1]<<1)
            insertion (I) ← (R[d-1]<<1)
            match (M) ← (oldR[d]<<1) | curPM
            R[d] ← D & S & I & M                     ▷ status bitvector for d errors
        if MSB of R[d] == 0, where 0 ≤ d ≤ k         ▷ check if MSB is 0
            startLoc ← i                             ▷ matching location
            editDist ← d                             ▷ found minimum edit distance
```

Bitap algorithm



```
Text Region:
CGTGA

Query Pattern:
CTGA

Edit Distance
Threshold (k):
1
```

```
PREPROCESSING  0
Pattern Bitmasks:
         CTGA
PM(A) = 1110
PM(C) = 0111
PM(G) = 1101
PM(T) = 1011

State Vectors:
   R0 = 1111
   R1 = 1111
```

```
Text[4]: CGTGA  1
oldR0 = 1111
oldR1 = 1111

R0 = (oldR0 << 1) | PM(A)
   = 1110

       D : oldR0                   = 1111
       S : oldR0 << 1              = 1110
R1 =   I : R0 << 1                 = 1100
       M : (oldR1 << 1) | PM(A)    = 1110
   = D & S & I & M                 = 1100
```

```
Text[3]: CGTGA  2
oldR0 = 1110
oldR1 = 1100

R0 = (oldR0 << 1) | PM(G)
   = 1101

       D : oldR0                   = 1110
       S : oldR0 << 1              = 1100
R1 =   I : R0 << 1                 = 1010
       M : (oldR1 << 1) | PM(G)    = 1101
   = D & S & I & M                 = 1000
```

```
Text[2]: CGTGA  3
oldR0 = 1101
oldR1 = 1000

R0 = (oldR0 << 1) | PM(T)
   = 1011

       D : oldR0                   = 1101
       S : oldR0 << 1              = 1010
R1 =   I : R0 << 1                 = 0110
       M : (oldR1 << 1) | PM(T)    = 1011
   = D & S & I & M                 = 0000
```

Alignment Found @ Location=2

```
Text[1]: CGTGA  4
oldR0 = 1011
oldR1 = 0000

R0 = (oldR0 << 1) | PM(G)
   = 1111

       D : oldR0                   = 1011
       S : oldR0 << 1              = 0110
R1 =   I : R0 << 1                 = 1110
       M : (oldR1 << 1) | PM(G)    = 1101
   = D & S & I & M                 = 0000
```

Alignment Found @ Location=1

```
Text[0]: CGTGA  5
oldR0 = 1111
oldR1 = 0000

R0 = (oldR0 << 1) | PM(C)
   = 1111

       D : oldR0                   = 1111
       S : oldR0 << 1              = 1110
R1 =   I : R0 << 1                 = 1110
       M : (oldR1 << 1) | PM(C)    = 0111
   = D & S & I & M                 = 0110
```

Alignment Found @ Location=0

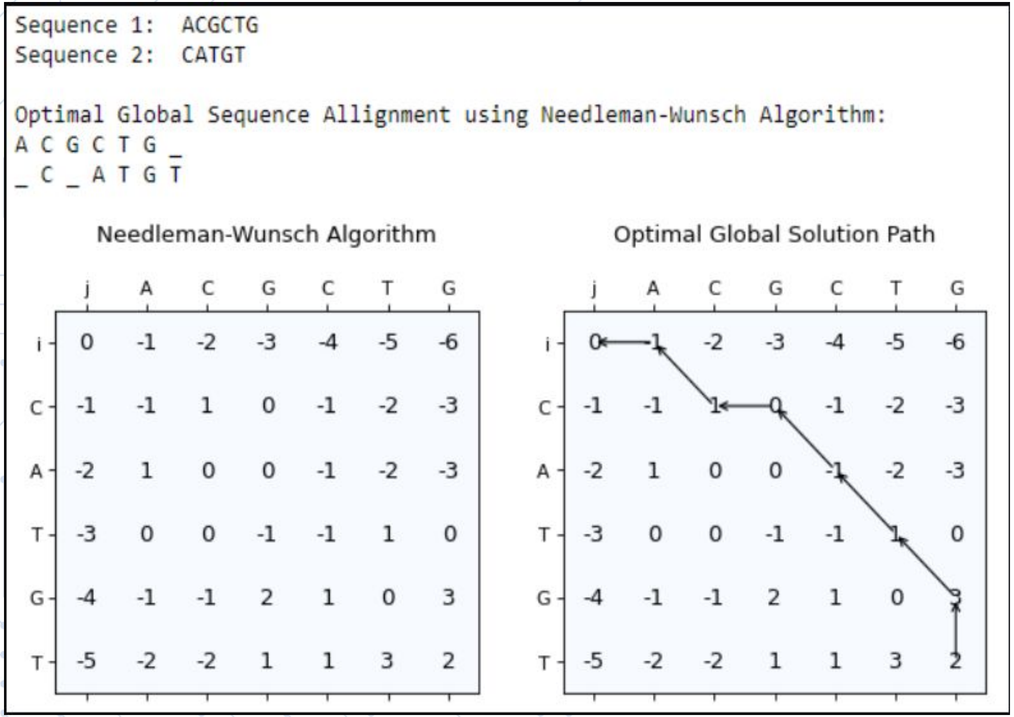
Needleman-Wunsch Algorithm

Termination : Bottom right

$$F_{ij} = max \begin{cases} F(\nwarrow) + S(match/mismatch) \\ F(\uparrow) + S(indel) \\ F(\leftarrow) + S(indel) \end{cases}$$

Scoring table:

Match	+1
Mismatch	-1
Indel	-1



```
Sequence 1:  ACGCTG
Sequence 2:  CATGT

Optimal Global Sequence Allignment using Needleman-Wunsch Algorithm:
A C G C T G _
_ C _ A T G T
```

Needleman-Wunsch Algorithm

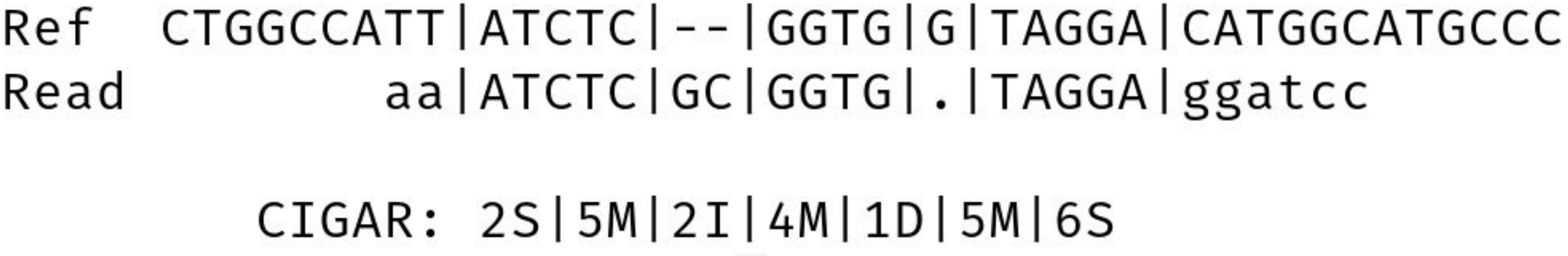
	j	A	C	G	C	T	G
i	0	-1	-2	-3	-4	-5	-6
C	-1	-1	1	0	-1	-2	-3
A	-2	1	0	0	-1	-2	-3
T	-3	0	0	-1	-1	1	0
G	-4	-1	-1	2	1	0	3
T	-5	-2	-2	1	1	3	2

Optimal Global Solution Path

	j	A	C	G	C	T	G
i	0	-1	-2	-3	-4	-5	-6
C	-1	-1	1	0	-1	-2	-3
A	-2	1	0	0	-1	-2	-3
T	-3	0	0	-1	-1	1	0
G	-4	-1	-1	2	1	0	3
T	-5	-2	-2	1	1	3	2

Traceback step

- Takes the TBM stored by the previous DC step and the output is alignment score along with a CIGAR string



```
Ref   CTGGCCATT|ATCTC|--|GGTG|G|TAGGA|CATGGCATGCCC
Read         aa|ATCTC|GC|GGTG|.|TAGGA|ggatcc

        CIGAR: 2S|5M|2I|4M|1D|5M|6S
```

How long does Read Mapping take?[1]

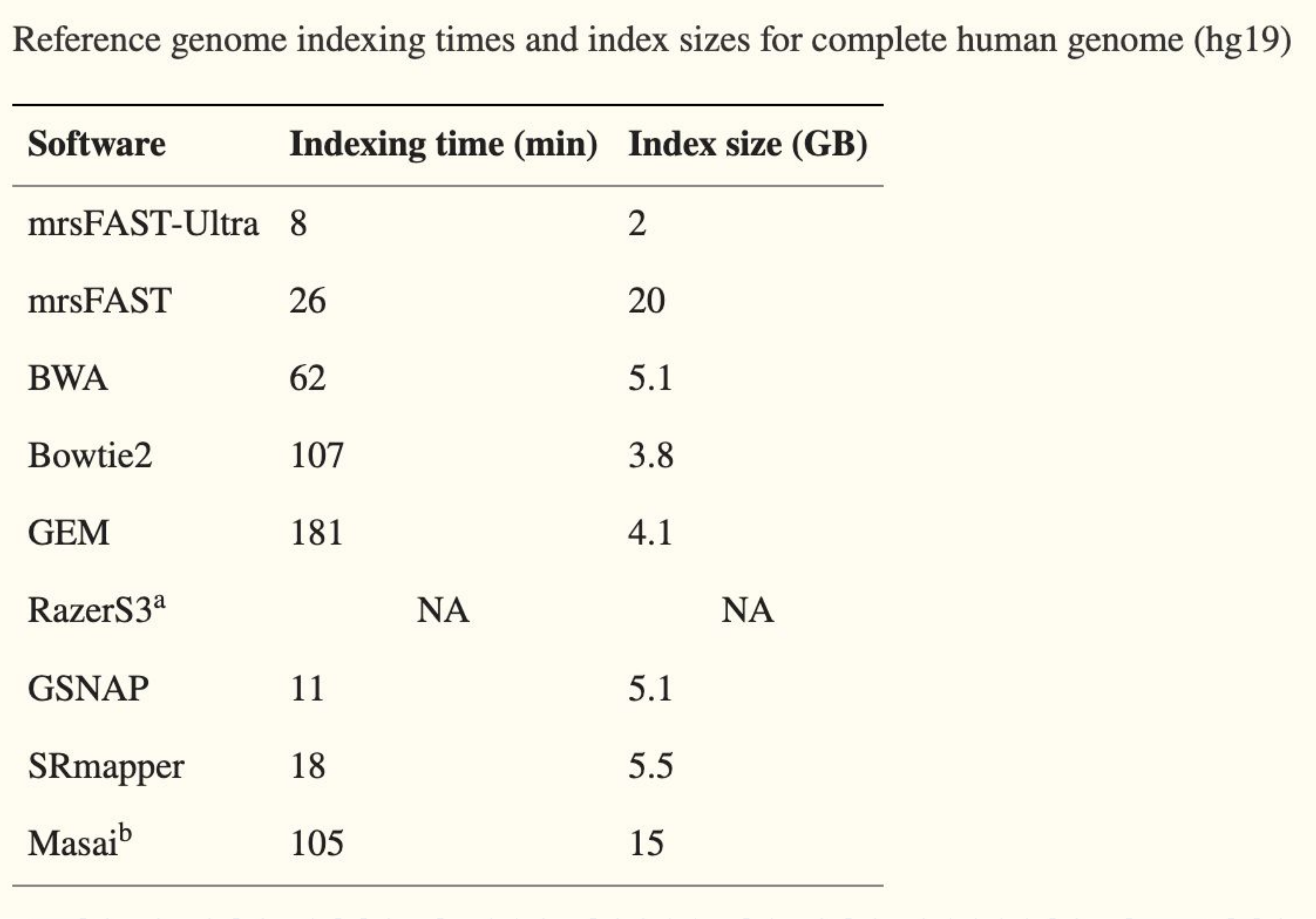
- The ASM performed during read mapping typically uses a computationally-expensive dynamic programming (DP) algorithm.
- This time consuming algorithm has long been a major bottleneck in the entire genome analysis pipeline, accounting for over 70% of the execution time of read mapping[1]

- In read mapping, there's three steps:
  - Indexing
  - Pre-alignment filtering
  - Sequence alignment
- Following Amdahl's law of speed up, we wish to make the sequence alignment step faster

Centre for
Heterogeneous and
Intelligent
Processing
Systems

Indexing[2]

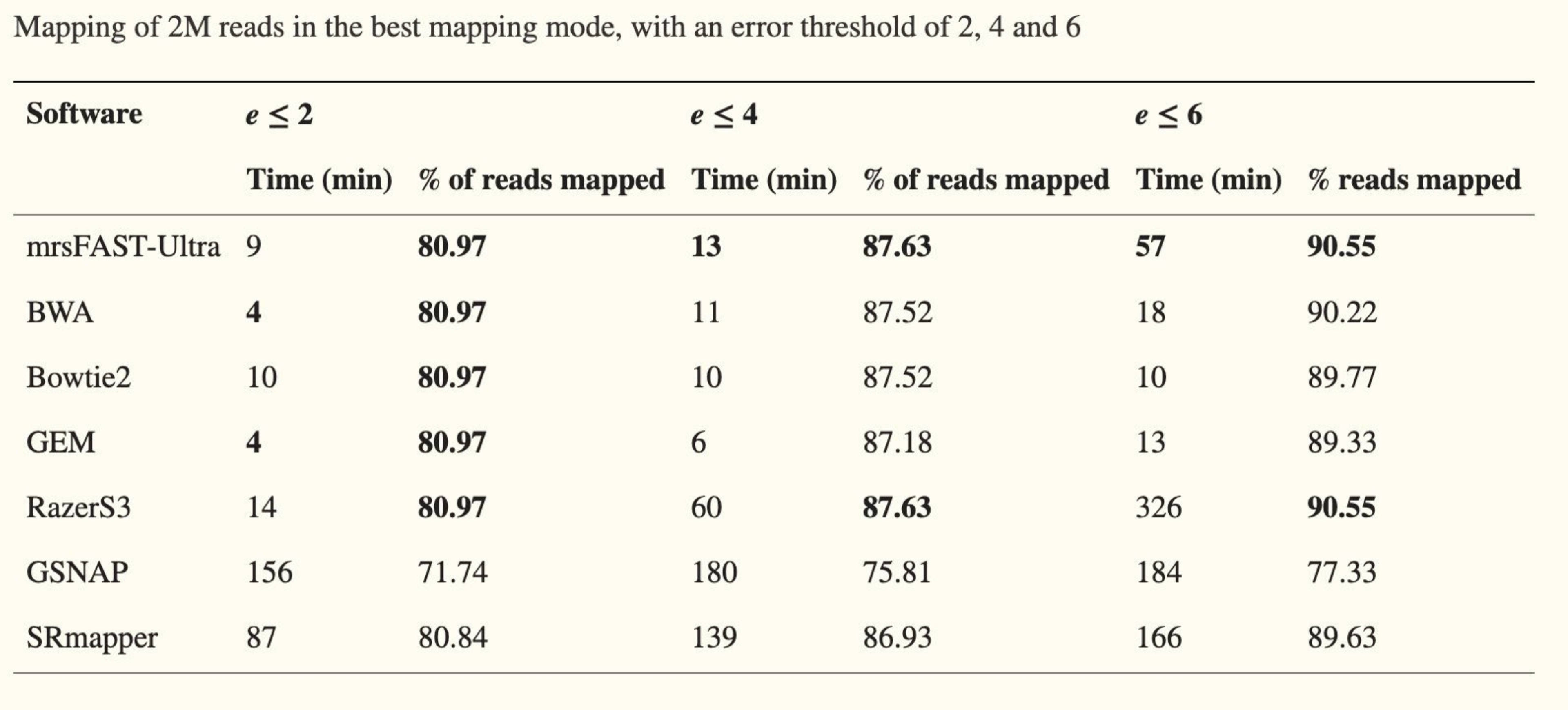
- The complete human genome (hg19) is 3,137,161,264 bp (or) characters long



Reference genome indexing times and index sizes for complete human genome (hg19)

Software	Indexing time (min)	Index size (GB)
mrsFAST-Ultra	8	2
mrsFAST	26	20
BWA	62	5.1
Bowtie2	107	3.8
GEM	181	4.1
RazerS3[a]	NA	NA
GSNAP	11	5.1
SRmapper	18	5.5
Masai[b]	105	15

Read-mapping[2]



Mapping of 2M reads in the best mapping mode, with an error threshold of 2, 4 and 6

Software	$e \leq 2$		$e \leq 4$		$e \leq 6$	
	Time (min)	% of reads mapped	Time (min)	% of reads mapped	Time (min)	% reads mapped
mrsFAST-Ultra	9	**80.97**	**13**	**87.63**	**57**	**90.55**
BWA	**4**	**80.97**	11	87.52	18	90.22
Bowtie2	10	**80.97**	10	87.52	10	89.77
GEM	**4**	**80.97**	6	87.18	13	89.33
RazerS3	14	**80.97**	60	**87.63**	326	**90.55**
GSNAP	156	71.74	180	75.81	184	77.33
SRmapper	87	80.84	139	86.93	166	89.63

Acceleration of Indexing[1]

- The indexing operation generates a table that is indexed by the contents of a seed, and identifies all locations where the seed exists in the reference genome.
- Indexing needs to be done only once for a reference genome, and eliminates the need to perform ASM across the entire genome.
- Seed from a read is taken from table, corresponding locations are used for ASM as only they can match entire read. Choosing the correct seed is the challenge: too short and multiple iterations are needed, too long and errors are high.

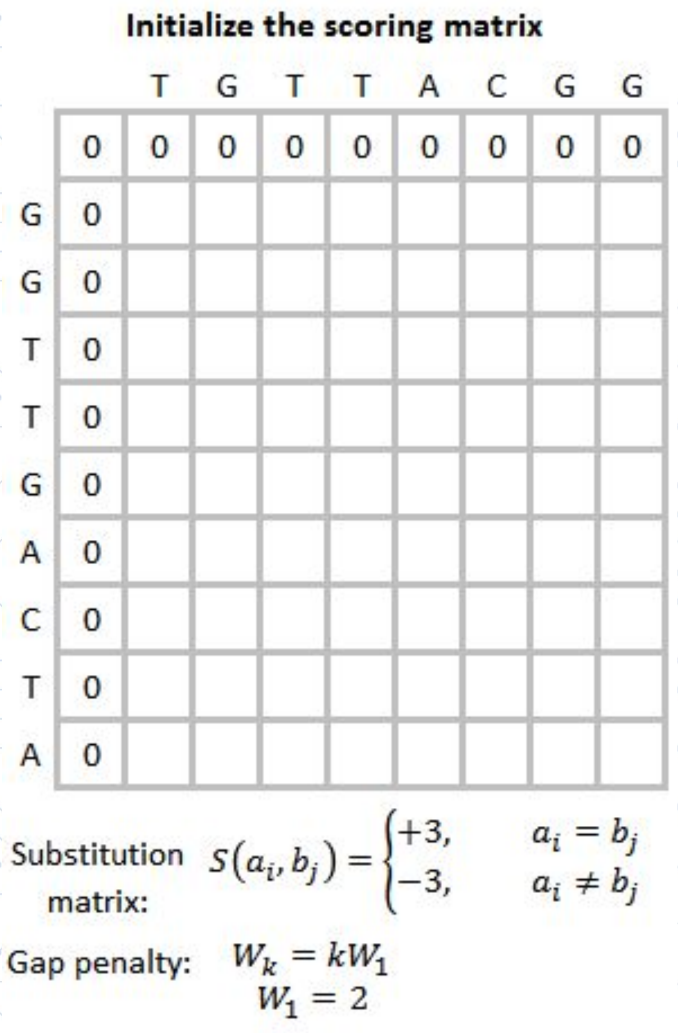
Read Mapper	**Platform**	**Novelty**	**Speed up**
MiniMap2 + FM Index	CPU	Reduces no. of seeds by finding best representative seeds from a group of adjacent seeds within a genomic region. **+** FM-Index: compressed representation of the full-text index, while allowing for querying the index without the need for decompression	**1.5x** smaller memory footprint than MiniMap2
BWA-MEM 2	CPU	Uncompressed FM-index (*10x of compressed*)	**2x** speedup in query
RADAR	ASIC (PIM)	Stores Index in memory, enables querying the same index concurrently	**5114x** of CPU implementation

Pre-alignment Filtering: A Comparison of Performance

Title	**Platform**	**Novelty**	**Speed Up**
FastHASH	CPU	Adjacency filtering and cheap K-mer selection	**19x** over mrFAST, a software read mapper that does not use a pre alignment filter
SHD	SIMD on CPU (Intel SSE)	Uses shifted hamming distance to reject reads above a certain edit distance threshold	**3.6x** over prior SIMD-based filters such as SeqAn and **24x** faster than swps3 by leveraging efficient SIMD instructions on mrFAST Is slower than FastHASH in some cases* but gives lesser false positives
GateKeeper	GPU, FPGA (Xilinx VC709 over PCIe)	SHD on different platforms.	**130x - 215x** over SHD (100bp - 300bp) **279x** over mrFAST (300bp)
Shouji	FPGA	*Improves accuracy of SHD as it does’nt consider indel*	**25x** over SHD (250bp)
SneakySnake	CPU, GPU, FPGA	*Ability to serve as a universal genome pre-alignment filter that works efficiently across multiple types of hardware (CPUs, GPUs, and FPGAs).*	Up to **5.4x** on CPUs over the state-of-the-art CPU-based pre-alignment filters Up to **18.4x** on GPUs over existing GPU-based pre-alignment filter Up to **5.5x** on FPGA over FPGA-based solutions
GenCache	PIM	Integration of Computation and Memory	up to **198x** over traditional CPU-based aligners and **7.9x to 11.2x** over GPU-based aligners

Acceleration of Alignment Stage

Title	Platform	Novelty	Speed up against
Parsail	CPU	SIMD support and can be used in multi core platforms	Up to **3.5x** over SSW
RAPID	PIM	eliminates the need for extensive data movement between memory and the CPU for fetching DP matrix values	Up to **15.6x** Over CPU-Based Implementations
Edilib	CPU	Use Myer's bit vector based algorithm which can be parallelised	Up to **10-100x** Speedup Over Naive Algorithms
GenASM	ASIC (PIM)	Custom hardware for approximate string matching	Up to **111x** Over CPU-Based Implementations Up to **15.8x** Over GPU-based accelerators
GWASBE	GPU		Up to **100x** Speedup Over CPU-Based GWAS Tools



Initialize the scoring matrix

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

Substitution matrix: $s(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

Gap penalty: $W_k = kW_1$
$W_1 = 2$

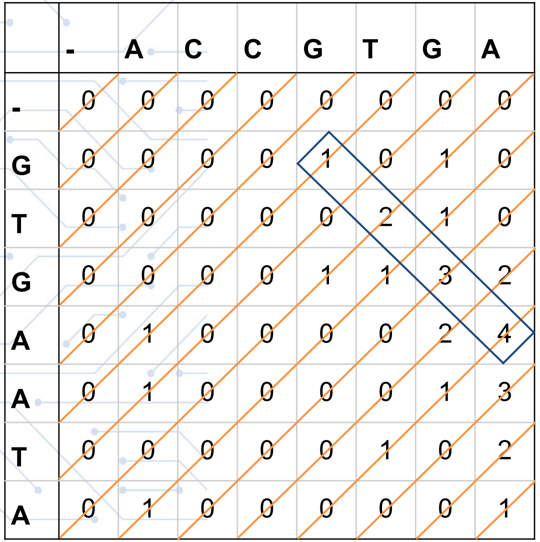
How does Smith Waterman work?

Anti diagonal parallelism

$$F_{ij} = max \begin{cases} F(i-1,\ j-1) + S(match/mismtach) \\ F(i,\ j-1) + S(indel) \\ F(i-1,\ j) + S(indel) \\ 0 \end{cases}$$

- Elements in in anti diagonal have no data dependency and can be computed in parallel

- Concurrently they’re dependent only on the anti diagonal above

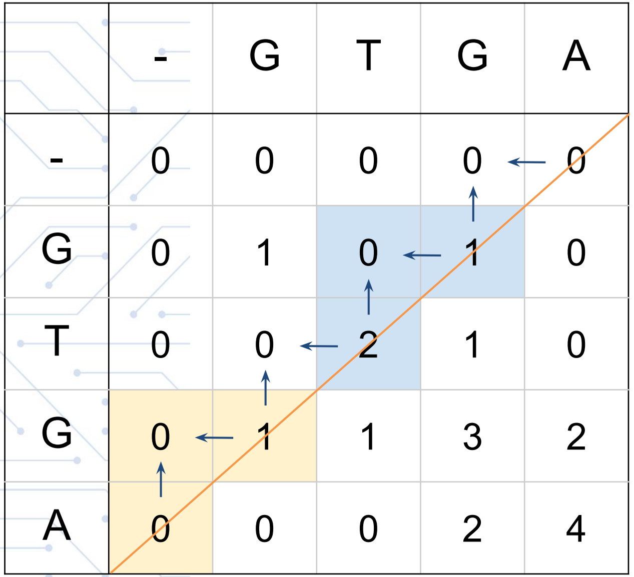


	-	A	C	C	G	T	G	A
-	0	0	0	0	0	0	0	0
G	0	0	0	0	1	0	1	0
T	0	0	0	0	0	2	1	0
G	0	0	0	0	1	1	3	2
A	0	1	0	0	0	0	2	4
A	0	1	0	0	0	0	1	3
T	0	0	0	0	0	1	0	2
A	0	1	0	0	0	0	0	1

Example to demonstrate

- The arrows show the elements that the anti-diagonal elements depend on. Since these elements are not dependent on elements in the same anti diagonal, they can be processed separately and assigned to different cores for parallel execution.

- The notable issue is that every adjacent anti-diagonal elements are accessing the same value of the matrix
  - For example, 0 & 1 access 0, 2 & 1 access 0



	-	G	T	G	A
-	0	0	0	0	0
G	0	1	0	1	0
T	0	0	2	1	0
G	0	1	1	3	2
A	0	0	0	2	4

Implementation approach

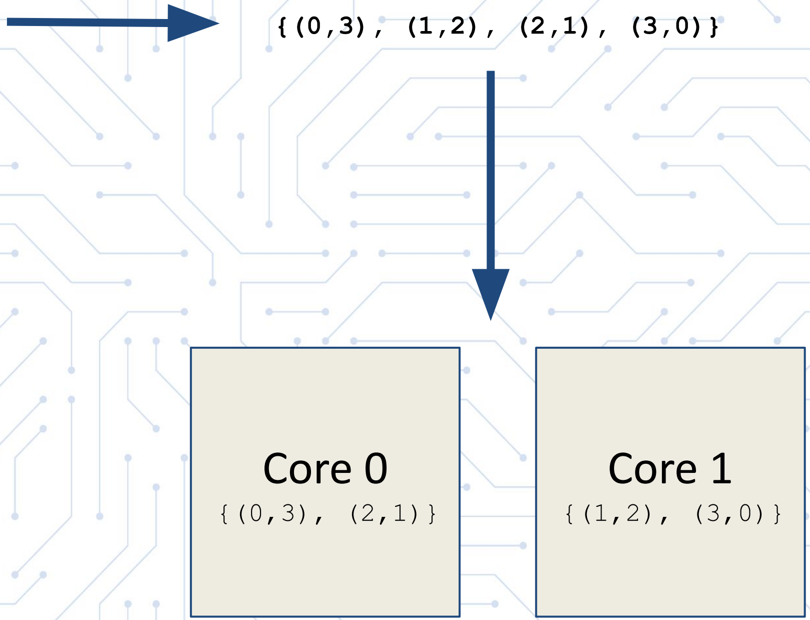
- Assume we have `p` cores and `k` elements in one anti diagonal
- Take the minimum of both (`min(p, k)`) so that `p <= k`
- Split computation based on core count equally, for computation of the anti-diagonal
- Synchronize via barrier b/w every anti-diagonal which has more than 1 element

EXAMPLE

1. 16 cores, 10 elements in an anti-diagonal, min(16,10) = 10 hence 10 cores assigned 1 element each to compute the anti diagonal.
2. 16 cores, 32 elements in anti-diagonal, min(16,32) = 16 hence 16 cores assigned 2 elements each to compute the anti-diagonal.

Very Basic Implementation visualised

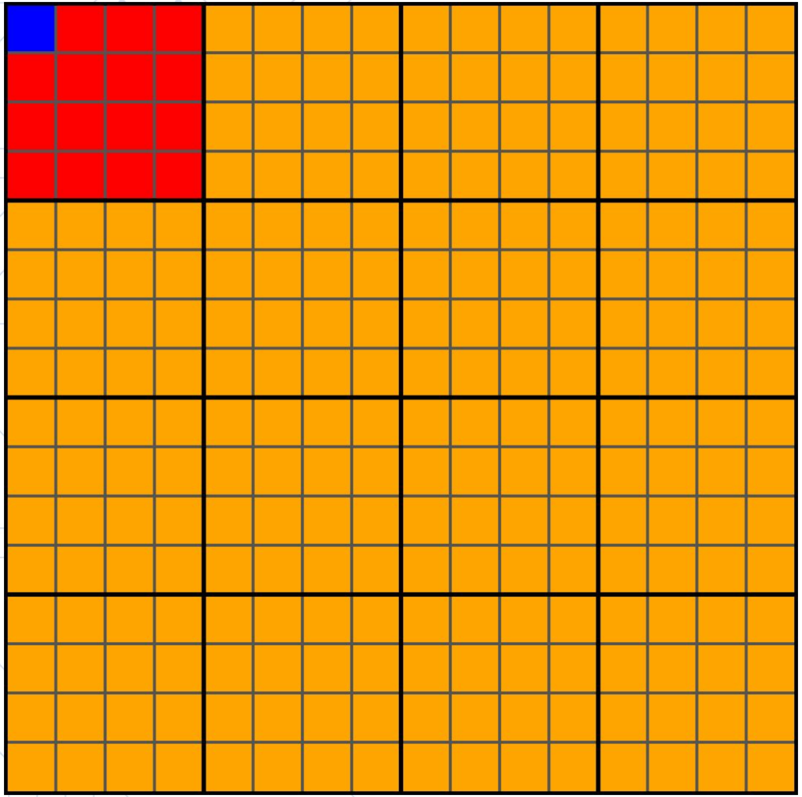
	-	X	X	X	X
-	0,0	1,0	2,0	3,0	4,0
X	0,1	1,1	2,1	3,1	4,1
X	0,2	1,2	2,2	3,2	4,2
X	0,3	1,3	2,3	3,3	4,3
X	0,4	1,4	2,4	3,4	4,4



```
{(0,3), (1,2), (2,1), (3,0)}
```

Division of Memory in Mempool[1]

- The reference and query string along with the DP matrix is stored in ScratchPad Memories (SPMs)
- We take an element of the DP matrix to be 2 bytes long, giving us a range of 0 to 65,536.
- One SPM(1KiB) can store 500 elements
- A tile has 16 such SPMs thus a tile can store 8,000 elements
- We divide the DP matrix elements into chunks of 8,000 elements and sequentially place it in all the SPMs in a tile
- 16 tiles can store a total of 128,000 elements
- Overall, <u>MemPool supports 1 MiB of SPM</u>, where we can store 512000 elements



DP matrix size calculation

Size of the DP matrix is calculated as follows:

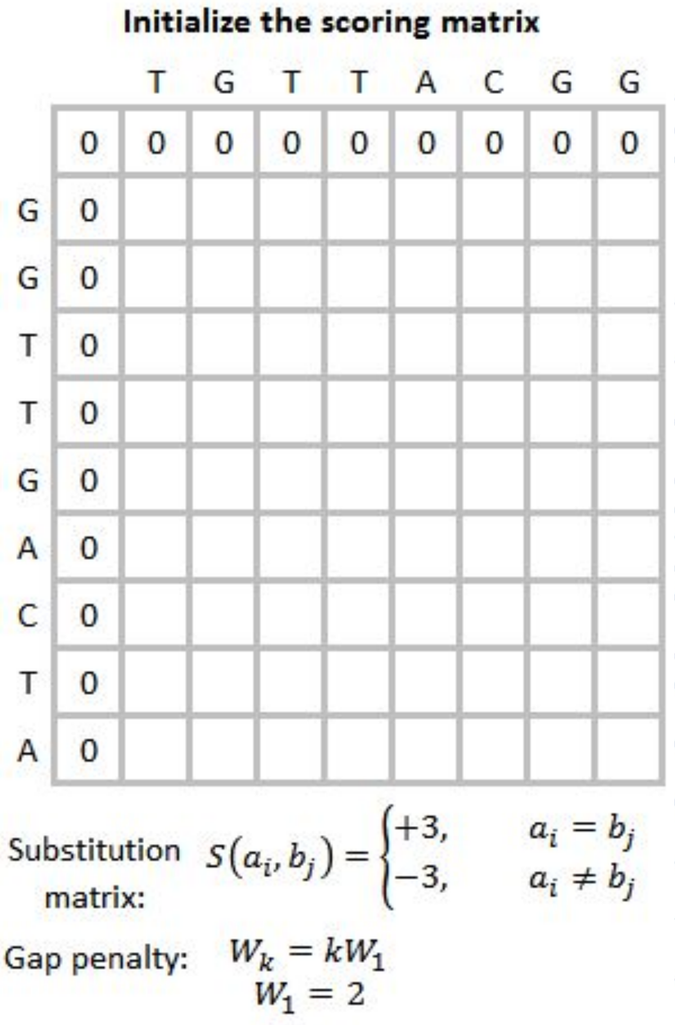
- The read genome is taken to be 300 base pairs, as a standard base-caller Illumina generates reads of length 300 bp.
- The reference genome is taken larger than the read size, approximately 400-500 base pairs
- The matrix has `(m+1)(n+1)` elements, where m is the size of the read, and n is the size of the reference genomes
- The +1 exists as the first row and column of the matrix is padded with zeros for the computation of the elements

	-	X	X	X	X
-	0	0	0	0	0
X	0	0	0	1	
X	0	0	2		
X	0	1			
X	0				

Platform constraints

- Every snitch core has a port to the interconnect
- Issue rate of snitch cores is 1 mem_req/cycle by a core
- Snitch core can have 8 outstanding memory requests in flight at any time
- We can make 194 concurrent mem_requests/cycle after which the interconnect is congested thus max cores we can use is 194

How does Smith Waterman work?



Initialize the scoring matrix

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

Substitution matrix: $s(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

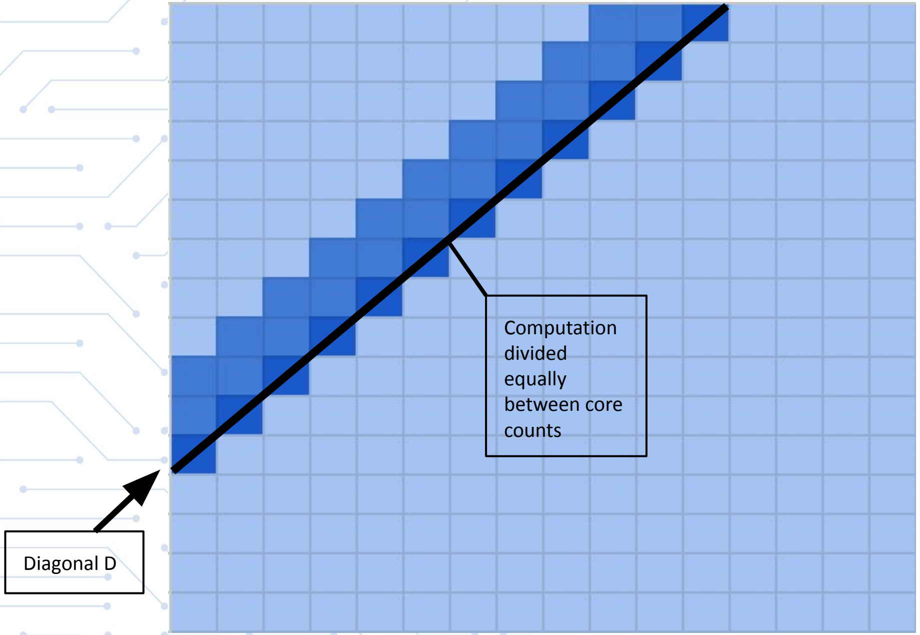
Gap penalty: $W_k = kW_1$
$W_1 = 2$

$$H_{(i,\ j)} = max \begin{cases} 0 \\ H_{(i,\ j-1)} + p \\ H_{(i-1,\ j)} + p \\ H_{(i-1,\ j-1)} \ + SbtCost \end{cases}$$

Initial approach to parallelisation

- We compute values in the same diagonal parallelly and then synchronize and move to the next diagonal

- We also vary number of active-cores according to length of the diagonal as initially elements in the diagonal increase and then decrease



Computation divided equally between core counts

Diagonal D

Issues with Default Smith-Waterman Implementation

- Data dependencies

$$H_{(i,\ j)} = max \begin{cases} 0 \\ \boxed{H_{(i,\ j-1)} + p} \\ \boxed{H_{(i-1,\ j)} + p} \\ \boxed{H_{(i-1,\ j-1)} \ + SbtCost} \end{cases}$$

	-	A	T
-	0	0	0
A	0	1	0

Highlighted cells are accessed by corresponding parts of the formula

Solution:

- The row comparison is done in an intermediate stage

$$H'_{(i,\ j)} = max \begin{cases} 0 \\ \boxed{H_{(i-1,\ j)} + p} \\ \boxed{H_{(i-1,\ j-1)} + SbtCost} \end{cases}$$

$$H_{(i,\ j)} = max \begin{cases} \boxed{H'_{(i,\ j)}} \\ \boxed{H_{(i,\ j-1)} + p} \end{cases}$$

Row-Parallel Implementation

$$\boxed{H'_{(i,\ j)}} = max \begin{cases} 0 \\ H_{(i-1,\ j)} + p \\ H_{(i-1,\ j-1)} + SbtCost \end{cases}$$

$$H_{(i,\ j)} = max \begin{cases} \boxed{H'_{(i,\ j)}} \\ H_{(i,\ j-1)} + p \end{cases}$$

Serial Comparison, requires previous value to be computed

↓

$$\boxed{H_{(i)}} = max_{1,2\ldots j} \begin{cases} \boxed{H'_{(i,\ j)}} \\ \boxed{H'_{(i,\ j-1)}} + p \\ \boxed{H'_{(i,\ j-2)}} + 2p \\ \boxed{H'_{(i,\ j-3)}} + 3p \\ \ldots \\ \boxed{H_{(i,\ 0)}} + jp \end{cases}$$

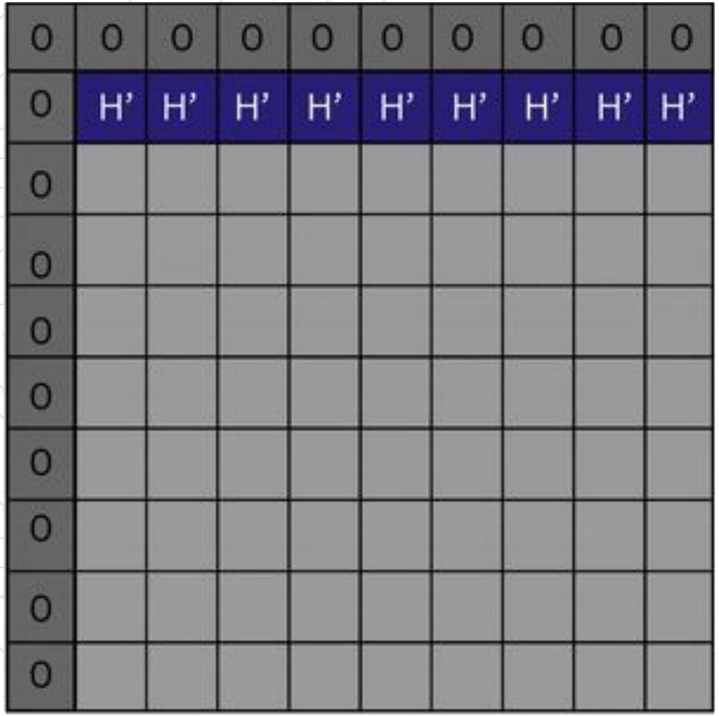
Reducing dependencies and computing entire row

↑

Using Prefix Max Scan

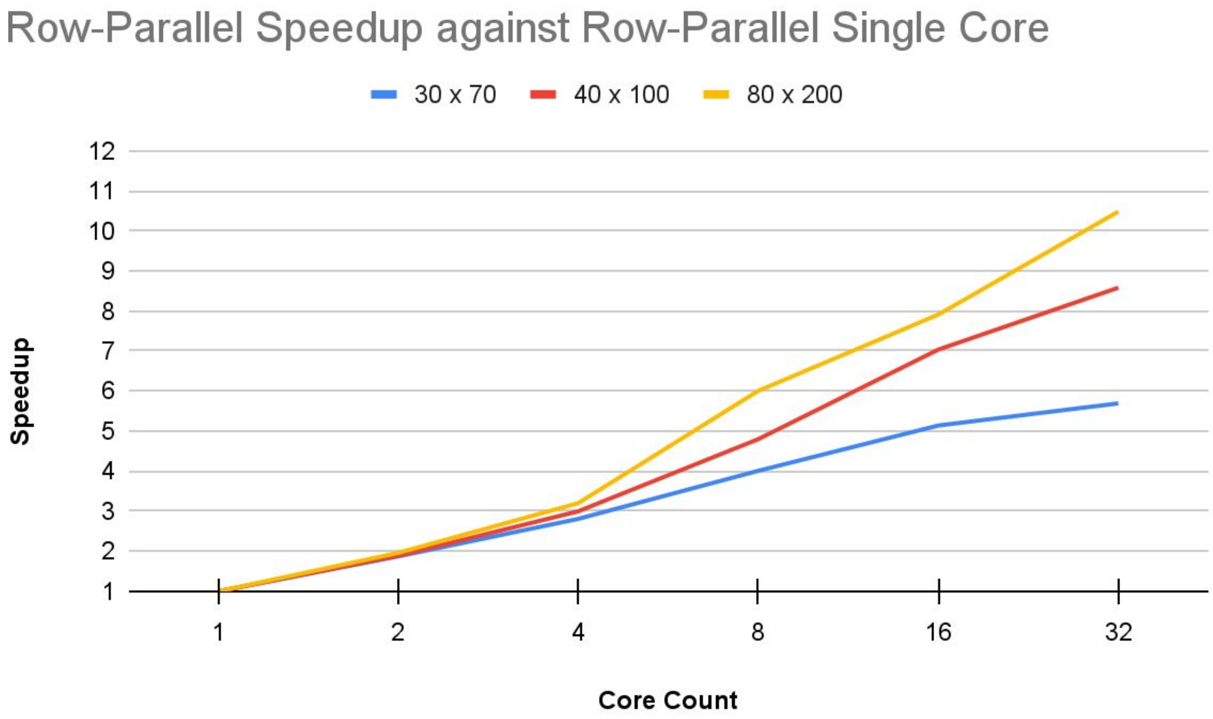
$$Input = [a,\ b,\ c,\ d]$$

$$Output = [max(a,a),\ max(a,b),\ max(a,b,c),\ max(a,b,c,d)]$$



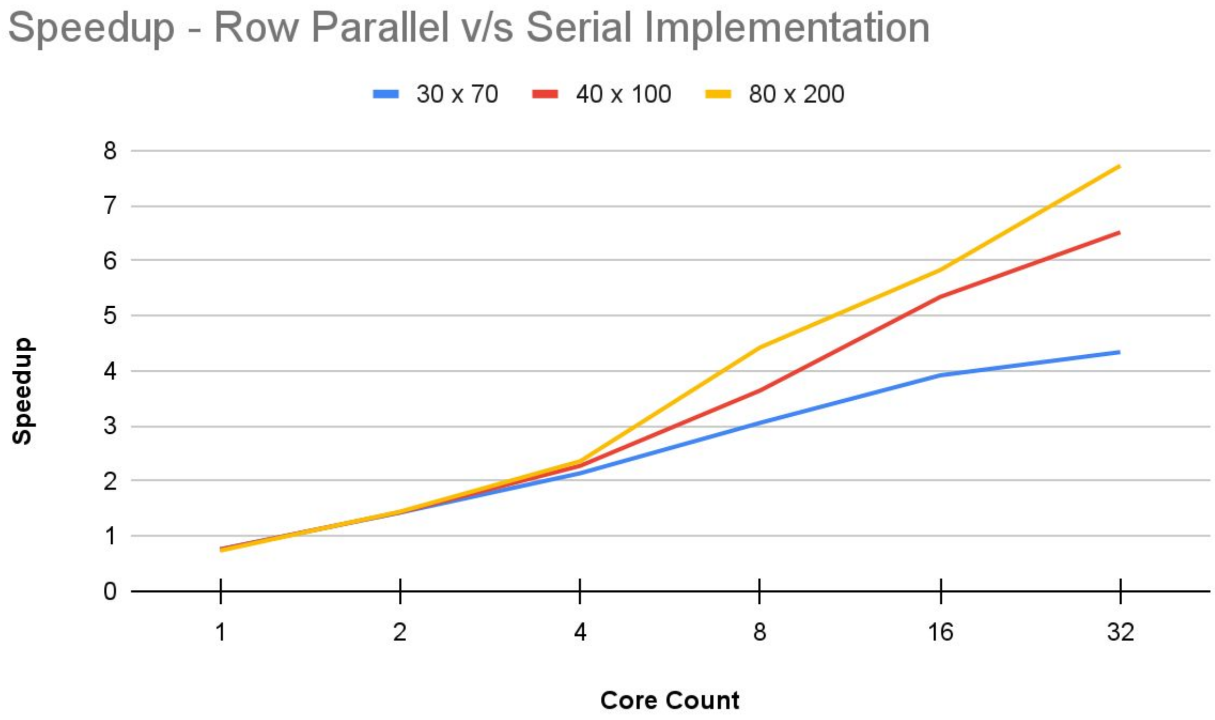
0	0	0	0	0	0	0	0	0	0
0	H’	H’	H’	H’	H’	H’	H’	H’	H’
0									
0									
0									
0									
0									
0									
0									
0									

Row-parallel Speedup



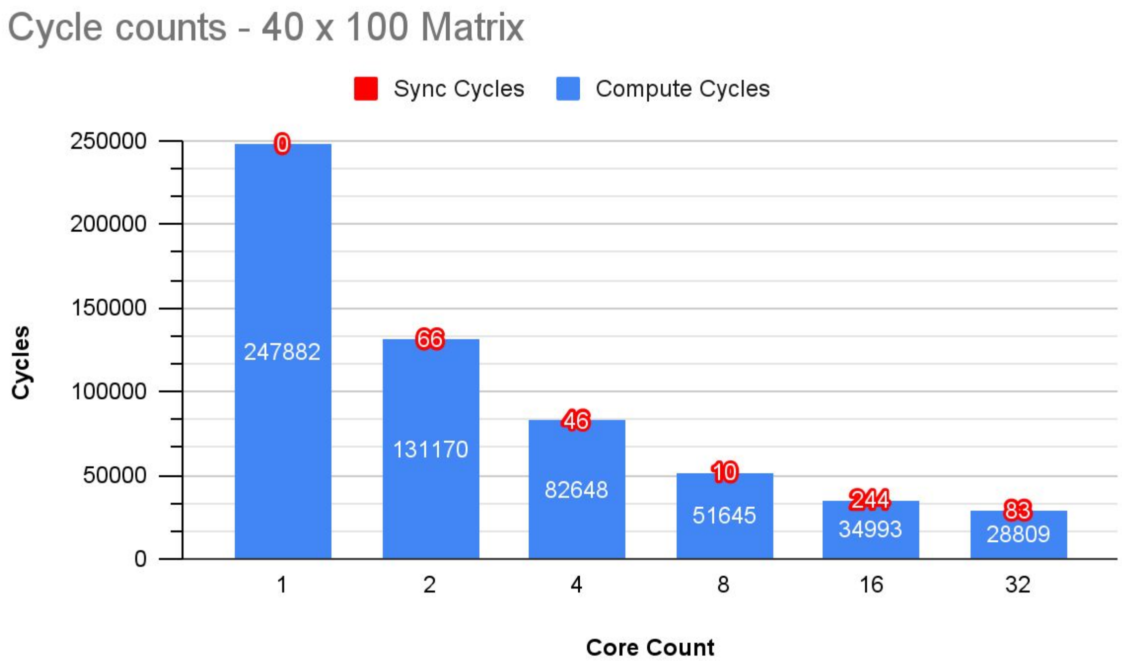
Row-Parallel Speedup v/s Unparallelised Smith-Waterman

Centre for Heterogeneous and Intelligent Processing Systems



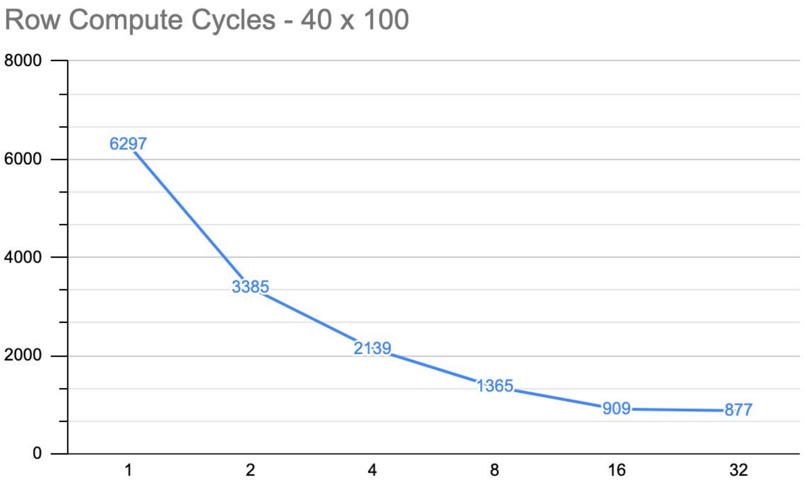
Row-Parallel Smith Waterman Cycles

Centre for **H**eterogeneous and **I**ntelligent **P**rocessing **S**ystems



Why no linear Speedup?

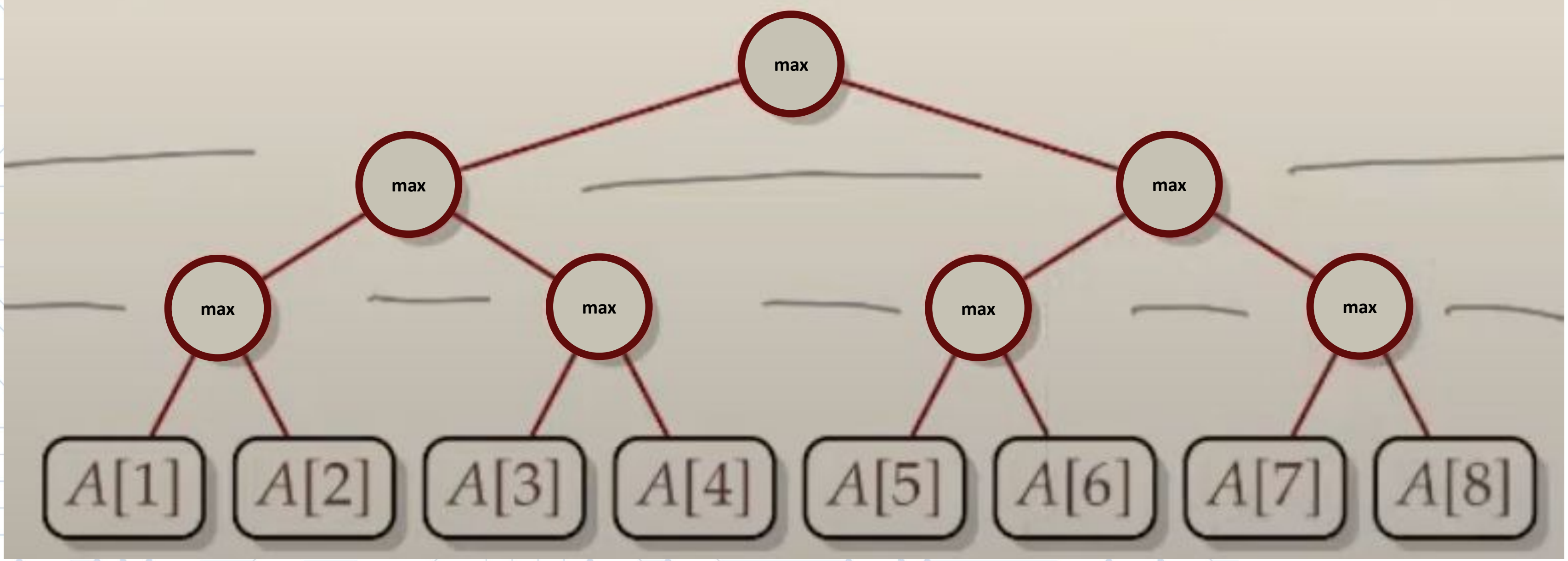
- Ideally, speedup in row compute time must become 2x when using twice the cores
- Speedup in row computation time from cc=1 to cc=2, cc=2 to cc=4 and so on is 1.8, 1.58, 1.56, 1.5, 1.03
- Elements per core is 100, 50, 25, 12, 6, 3 and for cc 1,2,4,8,16,32 and when the elements/core are very less the speedup from using twice the cores is negligible



The Competition - Other Implementations

- **Intel Manycore Co-processor:** Exploit the parallel processing capabilities of Intel's many-core architectures (Xeon Phi), to accelerate sequence alignment. For example: SWIMM
- **SIMD Vectorization:** Use Single Instruction Multiple Data (SIMD) instructions to speed up computation within threads. For example: SWIPE
- **GPU Acceleration:** Map the scoring matrix to GPU memory and compute diagonals or blocks using CUDA. For example: CUDASW++

How Parallel Prefix Max Scan works



Parallel Scan Implementation - An example

1) Calculate intermediate values ($H'_{(i,j)}$) for the 4 values [$O(1)$]

2) Use Prefix Max Scan to calculate intermediate array [$O(\log_2 n)$]

$$Input = [H'_{(1,\ 1)} + 3 * Indel,\ \ H'_{(1,\ 2)} + 2 * Indel,\ \ H'_{(1,\ 3)} + 1 * Indel,\ \ H'_{(1,\ 4)}]$$

$$= [a,\ \ b,\ \ c,\ \ d]$$

$$Output = [max(a, a),\ \ max(a, b),\ \ max(a, b, c),\ \ max(a, b, c, d)]$$

$$= [max(-5),\ \ max(-5,\ \ -4),\ \ max(-5,\ \ -4,\ \ -2),\ \ max(-5,\ \ -4,\ \ -2,\ \ 0)]$$

$$= [-5,\ \ -4,\ \ -2,\ \ 0]$$

	-	A	T	C	G
-	0	0	0	0	0
A	0				
G	0				
C	0				
G	0				

Match	+1
Mismatch	-1
Indel	-2

Parallel Scan Implementation - An example

3) Add result with the Indel-error array for $\mathrm{L}_{(i,j)}$ [O(1)]

$$Gap\ error\ array = [-(3*Indel),\ -(2*Indel),\ -(1*Indel),\ -(0*Indel)]$$

$$Gap\ error\ array = [+6,\ +4,\ +2,\ 0]$$

$$Output_{prefix\ max} = [-5,\ -4,\ -2,\ 0]$$

$$Result = L_{(i,\ j)} = [1,\ 0,\ 0,\ 0]$$

Match	+1
Mismatch	-1
Indel(p)	-2

4) Compare $\mathrm{L}_{(i,j)}$ with $\mathrm{H}_{(i,\ 0)}$ with appropriate Indel-errors [O(1)]

	-	**A**	**T**	**C**	**G**
-	0	0	0	0	0
A	0	1	0	0	0
G	0				
C	0				
G	0				

Time complexity: $O(m\log_2 n)$ for Matrix with m rows, n columns

Plan of action

- Split matrices into smaller units for more efficient computation
- Parallelize max scan efficiently on CPU
- Implement new parallel scan approach on Minpool platform
- Scale results on 256-core Mempool system
- Compare results with existing implementations

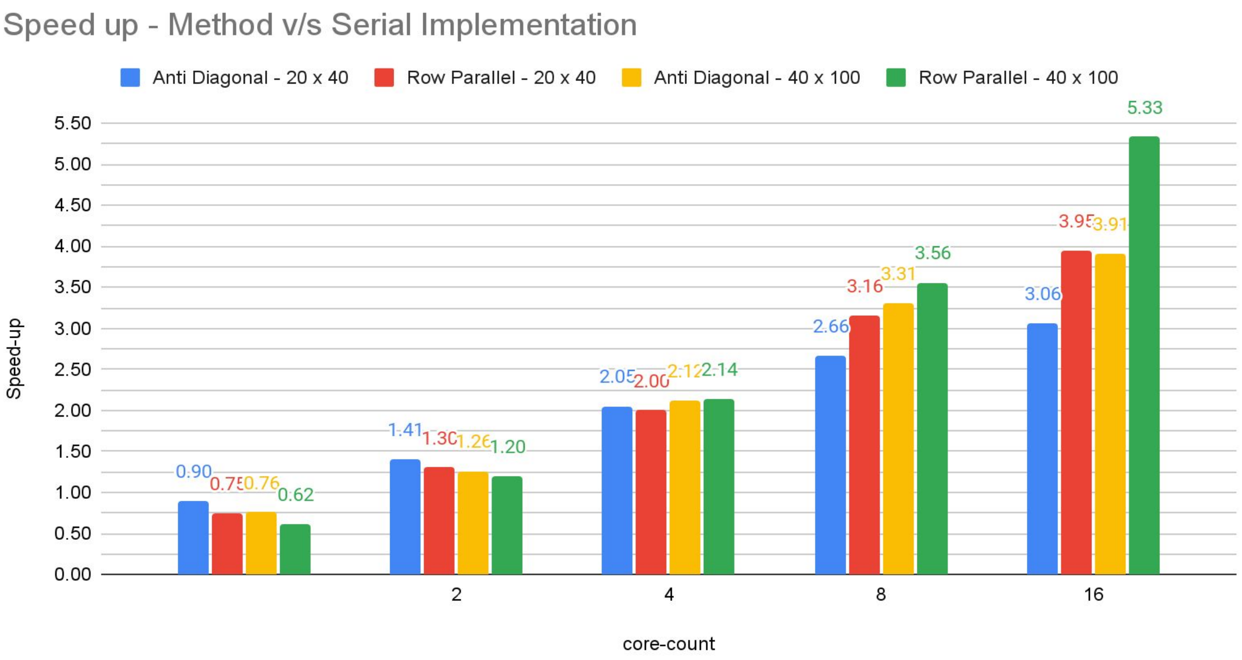
Fixing the AntiDiagonal error

- The compute_diagonal_segment function in AntiDiagonal kernel had two nested loops (outer: bound to 0, inner: 0 to bound), causing many wasted iterations as only diagonal elements are updated.
- When multiple cores are used, *all cores traverse the same loops* but *update different diagonal elements*, leading to redundant computations.
- Now every core calculates the start and end index of the portion it must compute and jumps to it, not performing redundant computation.

Parallel SW kernel vs serial single core implementation

- Comparison of parallel methods for the Smith-Waterman algorithm against a default, single-core, serial implementation

(Basically which algorithm is better)



Why RowParallel performs better than AntiDiagonal for 8 cores

- For `cc = 1 -> 4`, anti-diagonal outperforms row-parallel as the total kernel instructions of 14305 (AntiDiagonal), 17061 (RowParallel), and 9101 (Serial), but performance worsens as core count increases due to core stalls.

- Initial & final 7 diagonals cause 1 or more cores to stall, with 2516 cycles spent on these diagonals and 11542 cycles on the remaining diagonals, where all cores are utilized efficiently.

- AntiDiagonal and RowParallel take ~570-600 cycles to compute 40 elements using 8 cores, and optimizing initial/final diagonals (56 elements) to ~500 cycles could reduce AntiDiagonal cycles to ~12724, almost matching RowParallel's 12409 cycles

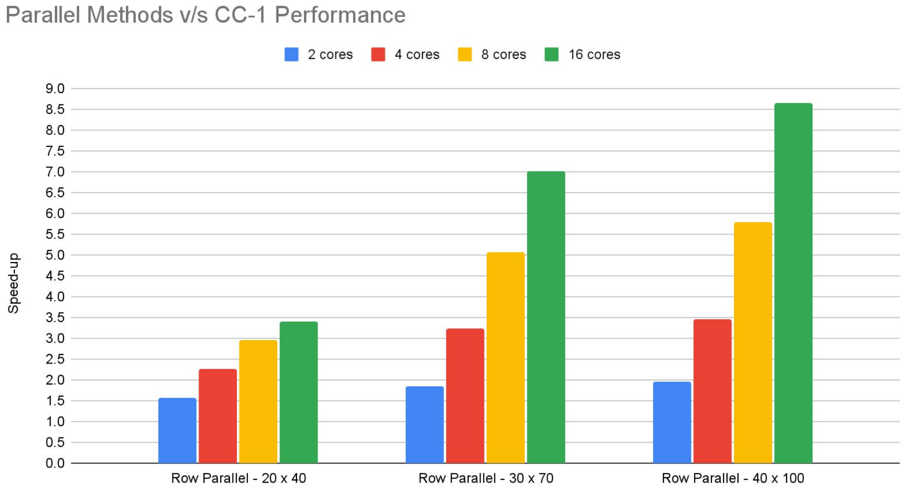
- Increasing core counts (e.g., $cc=16$) leads to more stalled cores across more diagonals (e.g., 0-14 diagonals with stalled cores), further degrading anti-diagonal performance.

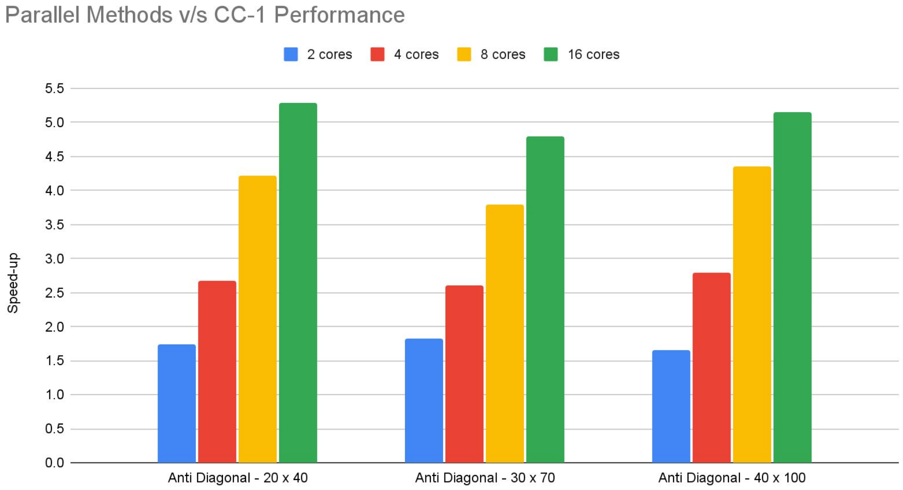
Why RowParallel performs better than AntiDiagonal for 8 cores

	RowParallel	AntiDiagonal
Total instructions	28873	14053
Total cycles	12409	14705
Cycles to compute rows/diagonals where 1+ cores are stalled	None(no stalled cores)	2516 {diagonals (0-6) & (52-59)} [56 elements]
Cycles to compute above elements using all 8 cores	None(no rows where cores are stalled)	535
Total cycles	12409	12724

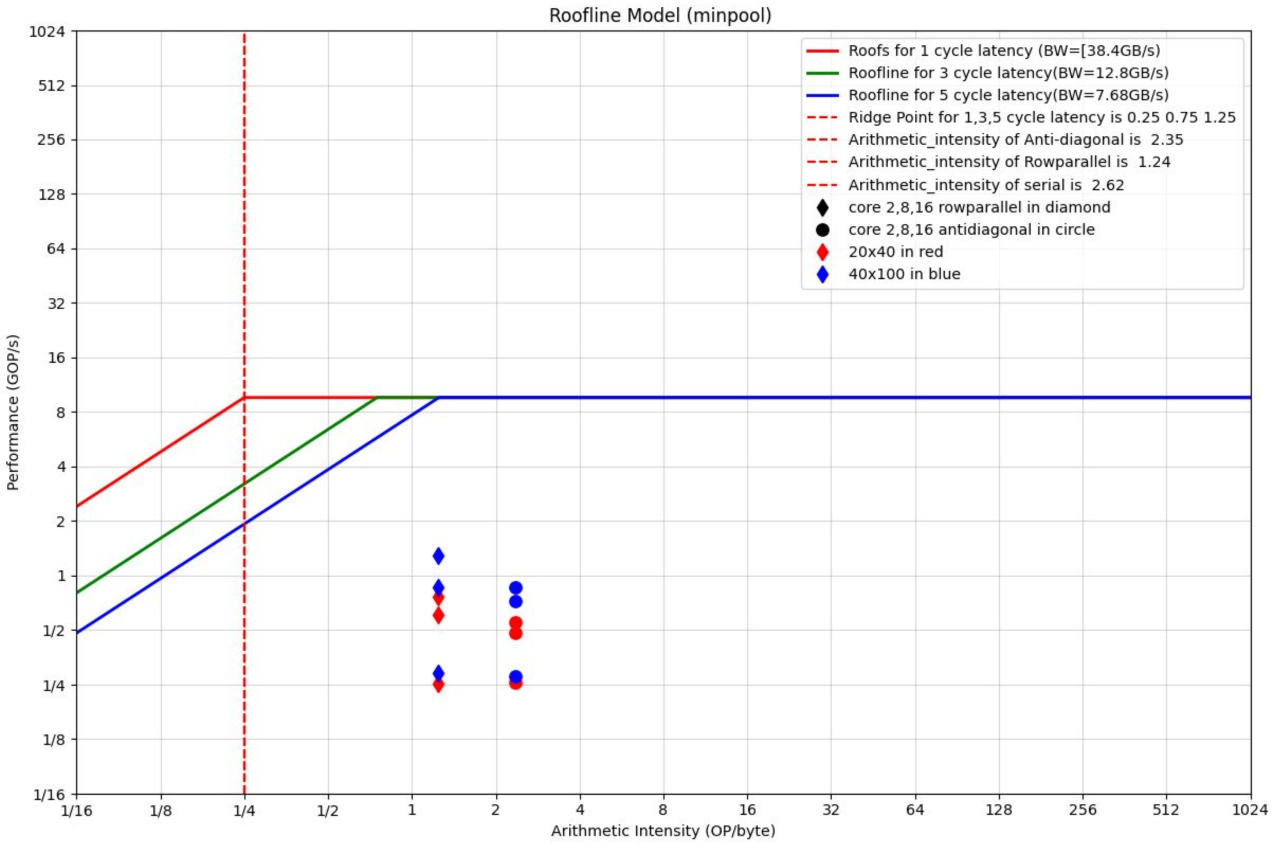
Centre for
Heterogeneous and
Intelligent
Processing
Systems

Speedup of Parallel Smith-Waterman Kernels: Single-Core vs. Multi-Core Execution





Roofline model

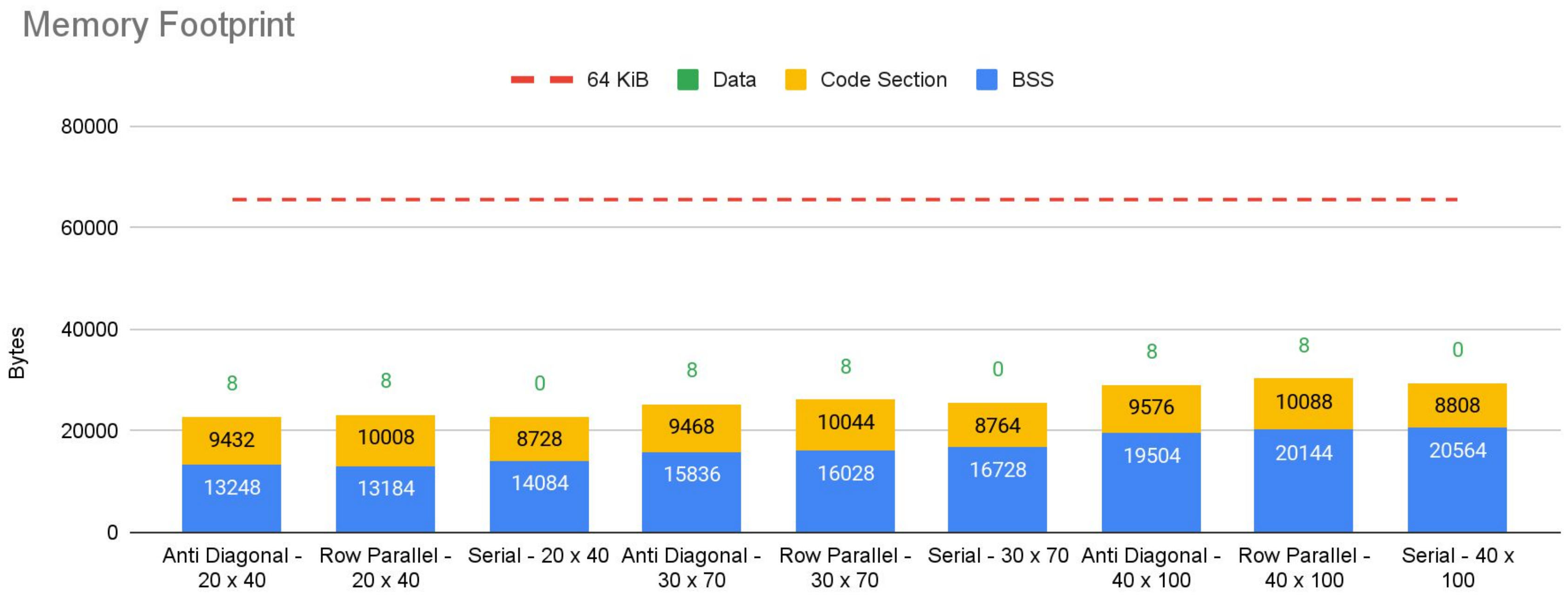


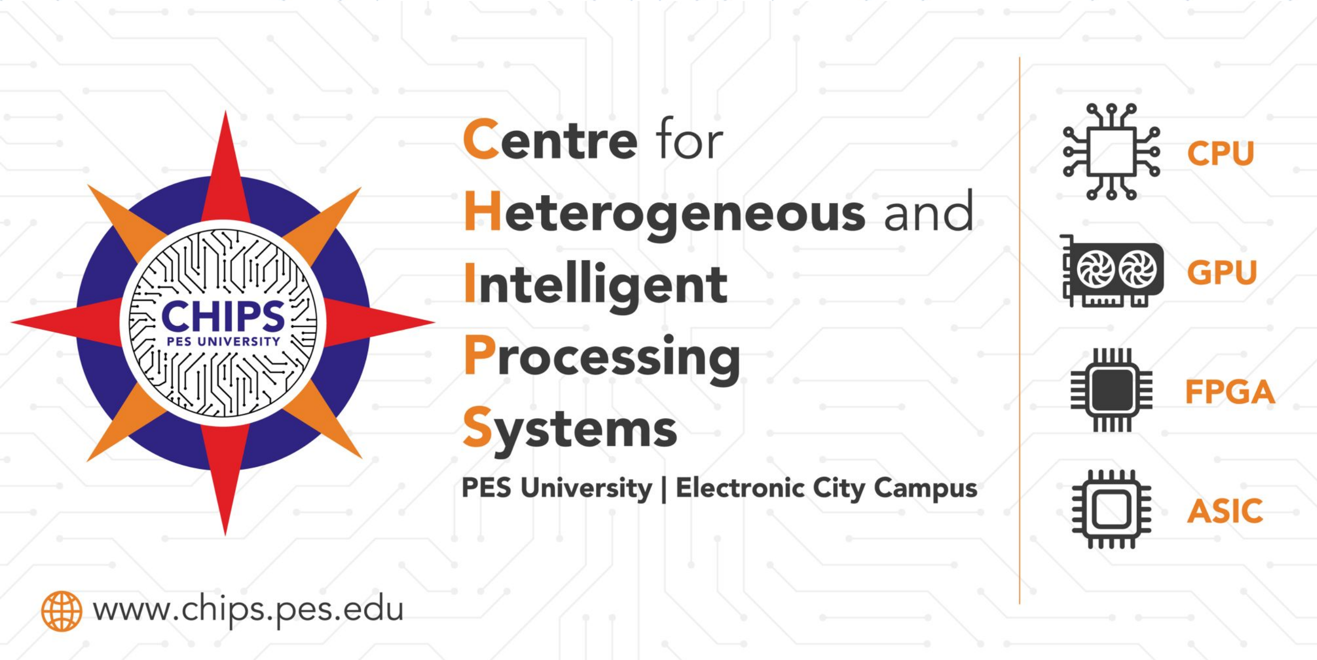
Centre for
Heterogeneous and
Intelligent
Processing
Systems

DIMENSION 20x40	anti-diagonal	rowparallel	serial
total memory instrunctions	2524	4491	1672
Total compute instr	11781	12570	6628
total instrunctions	14305	17061	8300
byte l/s	773	500	814
half-word l/s	1283	3196	831
word l/s	468	795	27
Total bytes required	5211	10072	2584
ACC stalls	712	1899	1125
LSU stalls	1260	914	814
arithmetic intenstiy	2.260794473	1.248014297	2.56501548

DIMENSION 40x100	anti-diagonal	rowparallel	serial
Total compute instr	54967	60518	32866
total memory instr	12535	22812	8217
total instr	67502	83330	41083
byte l/s	3953	2194	4062
half-word l/s	7427	17238	4118
word l/s	1155	3380	37
Total bytes required	23427	50190	12446
ACC stalls	3936	10524	5877
LSU stalls	6204	4493	4060
arithmetic intenstiy	2.346309813	1.205778043	2.640687771

Memory Footprint





Centre for
Heterogeneous and
Intelligent
Processing
Systems

PES University | Electronic City Campus

www.chips.pes.edu

Thank you