PORTING AND ADAPTING APPLICATIONS TO ARM'S SVE

Jonathan Beaumont, Dong-hyeon Park, Subhankar Pal, Trevor Mudge

Introduction

- Scalable Vector Extension (SVE)
- Extends Advanced SIMD (NEON) beyond 128 bit vectors
- NEON designed for DSP, media codecs etc.
  - Fixed vector sizes (128 bits)
  - Simple control flow
  - Regular, contiguous data structures
- SVE extends to HPC applications
  - Longer, per-implementation vector sizes (128-2048 bits)
  - Complex control flow
  - Irregular data structures

Scalable vectors

- Different PPA restrictions require distinct hardware vector lengths (e.g. mobile processors versus server hardware)

- SVE enables hardware implementations to choose vector length multiple of 128 bits (up to 2048)
  - Program code is agnostic; hardware manages automatically

```
ld1w z1.s, p1/z, [x1, x2,  lsl #2] ; load VL sized vector from
                                     memory location [x1+4*x2]

incw x2                            ; increment x2 counter by VL
whilelt p1.s, x2, x3               ; loop while x2 < x3
b.mi     loop
```

Per-lane predication

- Hardware dynamically masks inactive lanes
  - Support if/else statements
- Handles scalar loops which do not iterate an exact multiple of vector length
  - No fix-up code or strip-mining

```
while(i < end) {
  if(check[i])
   //do stuff
  i++;
}
```

```
whilelt p1.s, w11, w12
b.pl end_loop
loop:
  ld1w    z1.s, p1/z, [x4, x11,  lsl #2]
  cmpeq p2.s, p1/z, z1.s #0
  //do stuff [p2/z]
  incw    x11
  whilelt p1.s, w11, w12
  b.mi      loop
end_loop:
```

Vector load-stores

- HPC data structures often use complex data structures using pointers
- SVE supports gather-load and scatter-store instructions
  - Allows indirect-access to non-contiguous memory arrays within a single instruction

```
int new_val = arr[idxs[i]] + 1;
```

```
ld1w    z2.s, p1/z, [x5, x11,  lsl #2]
ld1w    z3.s, p1/z, [x1, z2.s, uxtw #2]
fadda  s13,  p1, s13, 1
```

Horizontal reductions

- Dependencies between elements in a vector can be resolved with special instructions
  - Summation, minimum, maximum, and logical reductions supported in single instructions

Example Sparse Matrix-Vector Multiplication:

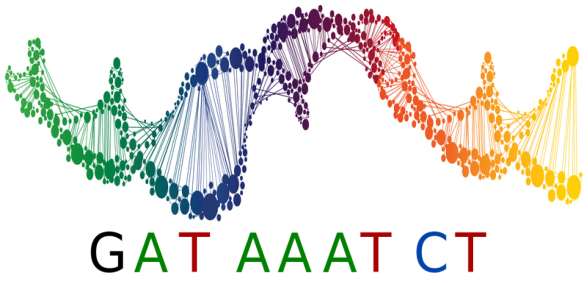
```
scvtf s13, xzr                              ; sum = 0
loop:
  ld1w    z1.s, p1/z, [x4, x11,  lsl #2]    ; ld vals[i]
  ld1w    z2.s, p1/z, [x5, x11,  lsl #2]    ; ld cols[i]

  ld1w    z3.s, p1/z, [x1, z2.s, uxtw #2]   ; ld vec[cols[i]]
  fmul    z1.s, p1/m, z1.s, z3.s            ; vals[i] * vec[cols[i]]
  fadda   s13,  p1, s13, z1.s               ; sum += sum(z1)
  incw    x11
  whilelt p1.s, w11, w12
  b.mi      loop
end_loop:
```

Target Workloads

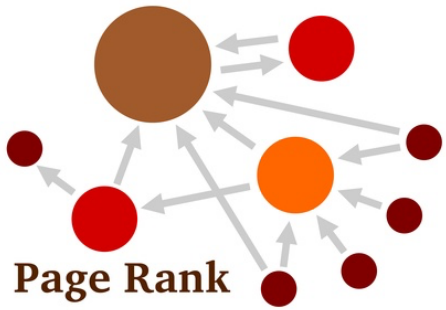
- Many scientific workloads make use of vector operations through linear algebra subroutines
- These workloads have wide range of sparsity in their structures, which affects
  - Control flow regularity
  - Memory access patterns
- We look at two case studies with complementary sparsity:

Target Workloads

- Genetic sequence alignment
  - Dense vectors
  - Fine grained parallelism
  - Moderate control irregularity
  - Moderate memory irregularity



- Graph analytics
  - Extremely sparse (1 in 10^6) matrices
  - Emphasis on effective data movement over minimizing computation
    - Employ outer-product on matrix multiplications
  - Little control divergence
  - Significant memory irregularity



Sequence Alignment

- Before analysis can be performed on genome fragments, they must be aligned to a reference gene

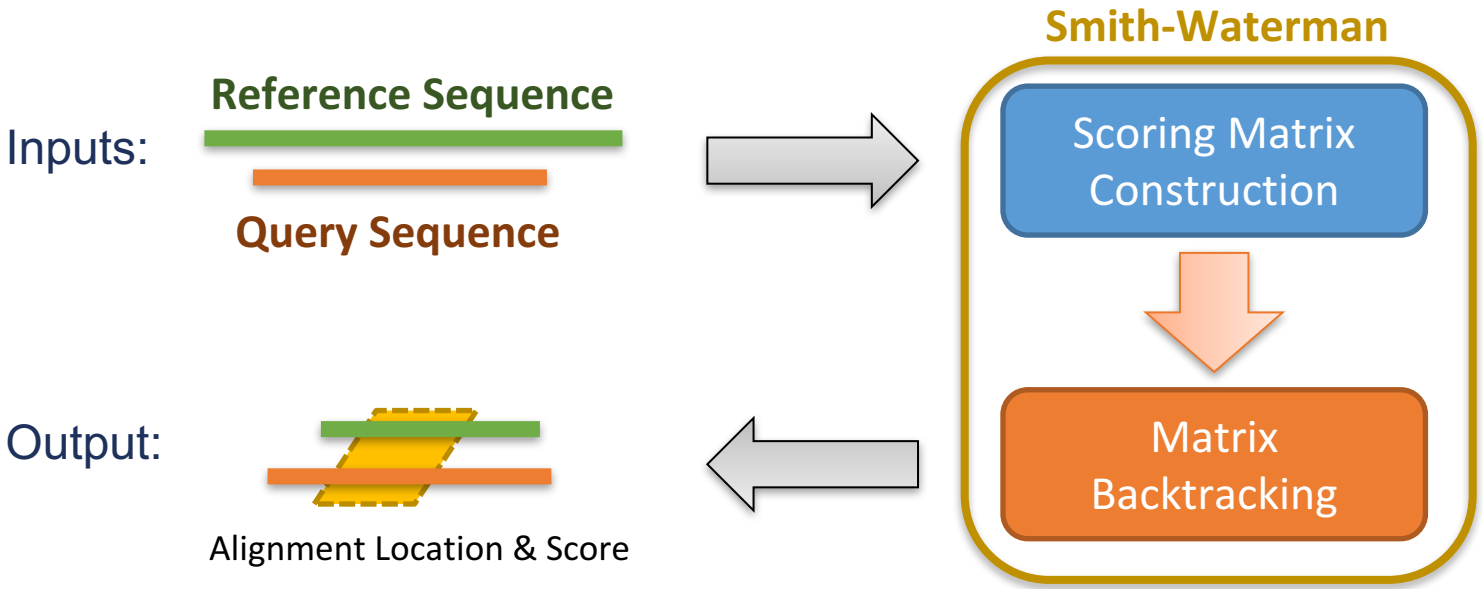
- Multiple alignments corresponding to various insertions, deletions, and offsets must be evaluated, leading to a dense, vectorized workload

- We evaluate the Smith-Waterman Algorithm

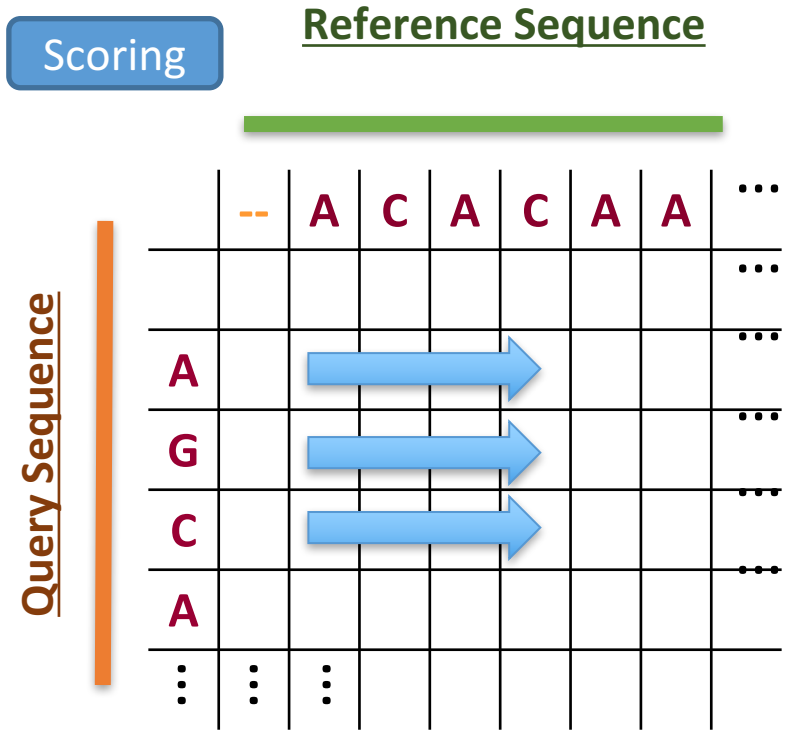
Smith-Waterman Algorithm

- Local sequence alignment algorithm developed in 1981
- Smaller amount of parallelism (100s of elements or less)

Local sequence alignment algorithm developed in 1981



Scoring Matrix Construction



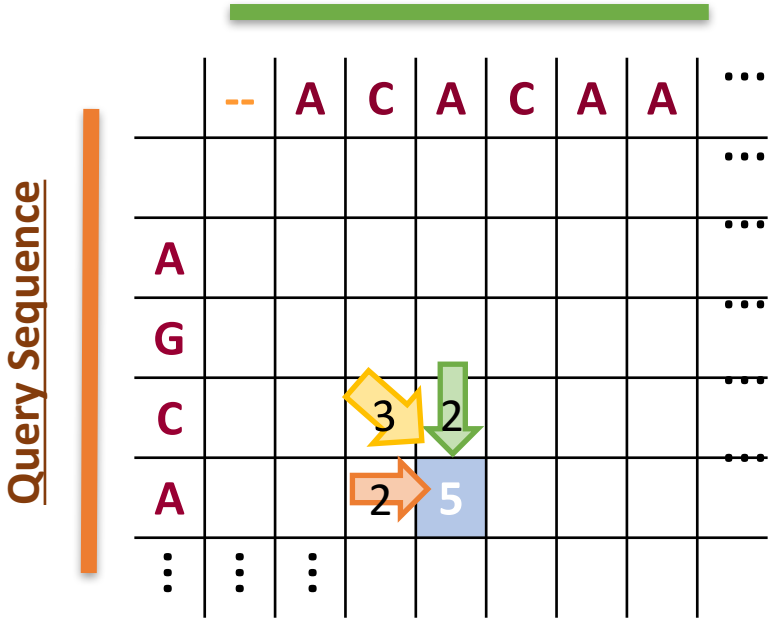
$$H(m,n) = max \begin{cases} E(m,n) \\ F(m,n) \\ H(m-1,n-1) + S(a_m, b_n) \end{cases}$$

$$E(m,n) = max \begin{cases} H(m,n-1) - g_o \\ E(m,n-1) - g_e \end{cases}$$

$$F(m,n) = max \begin{cases} H(m-1,n) - g_o \\ F(m-1,n) - g_e \end{cases}$$

Scoring Matrix Construction

Scoring

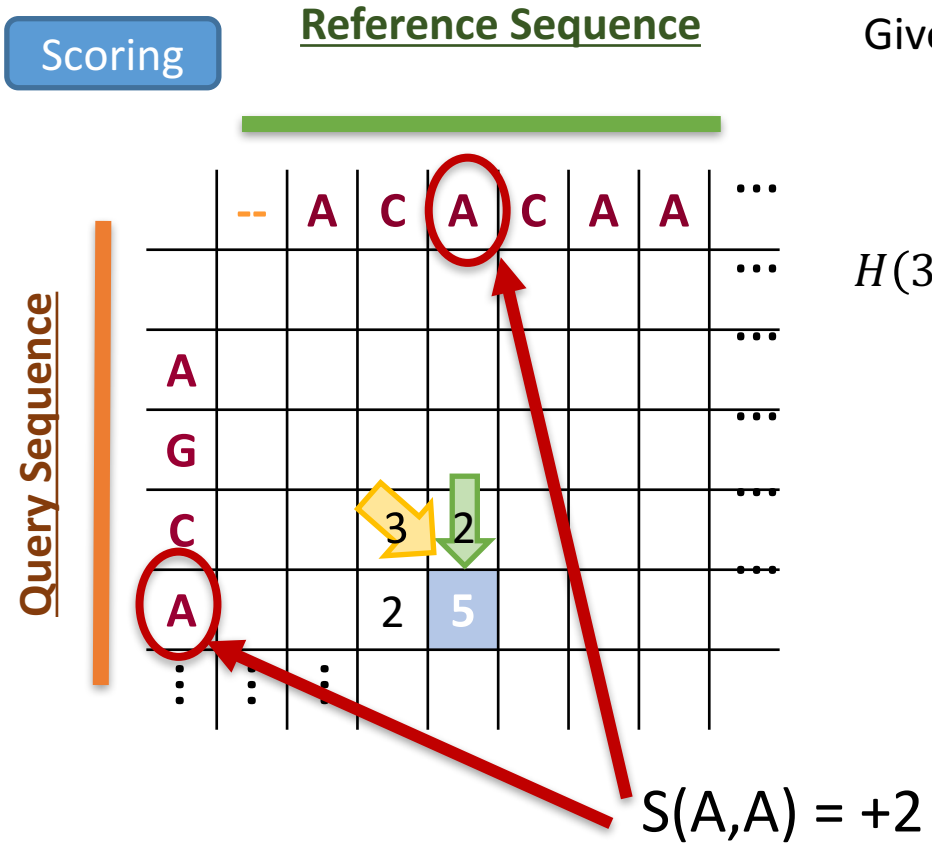


$$H(m,n) = max \begin{cases} E(m,n) \\ F(m,n) \\ H(m-1,n-1) + S(a_m, b_n) \end{cases}$$

$$E(m,n) = max \begin{cases} H(m,n-1) - g_o \\ E(m,n-1) - g_e \end{cases}$$

$$F(m,n) = max \begin{cases} H(m-1,n) - g_o \\ F(m-1,n) - g_e \end{cases}$$

Scoring Matrix Construction



Given:

$S(a, b) = +2$

Gap penalty = 1

$H(3,4) =$

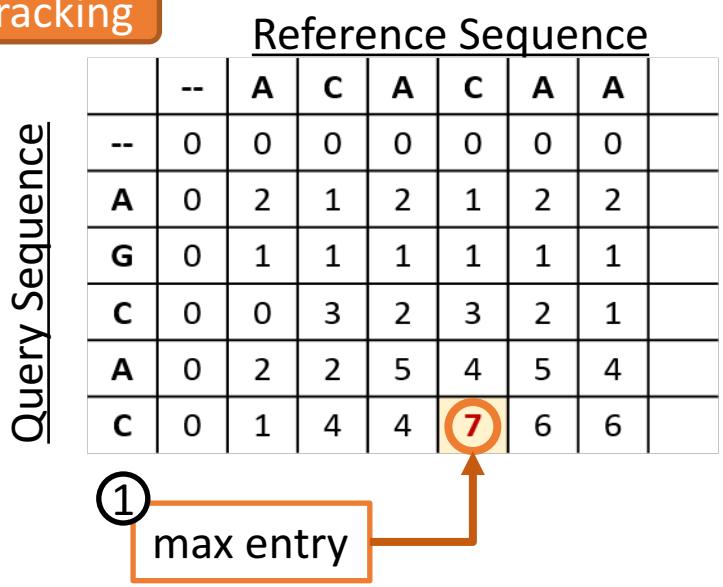
$$\max \begin{bmatrix} F(3,4) = 2 - 1 \\ E(3,4) = 2 - 1 \\ \mathbf{3 + 2 = 5} \end{bmatrix}$$

$H(3,4) = 5$

Backtracking

Finds the best local alignment from the scoring matrix

Backtracking

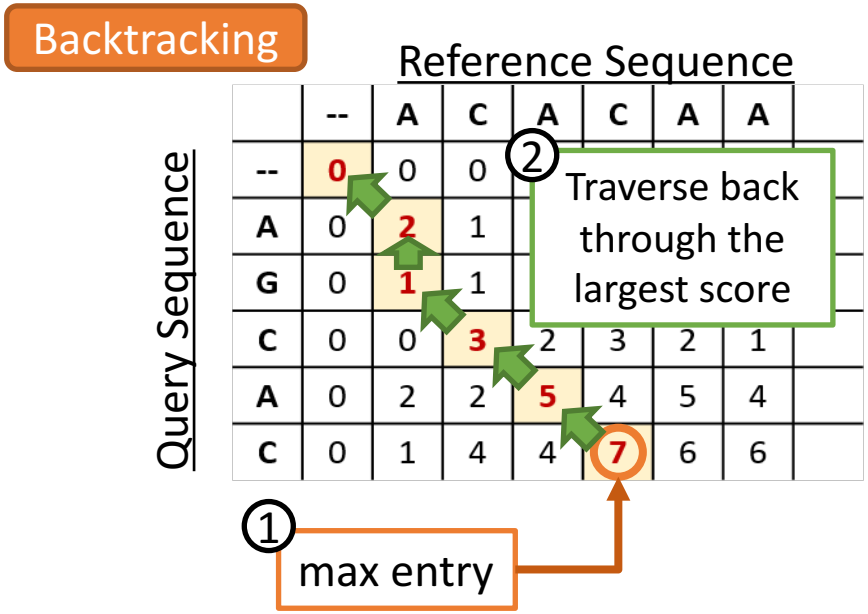


Step 1.

Search through the matrix and find the entry with the largest score

Backtracking

Finds the best local alignment from the scoring matrix



Step 2.

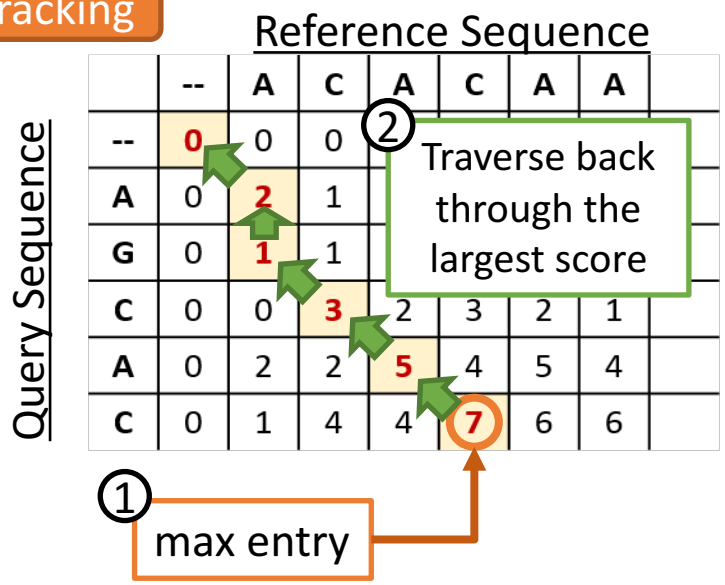
Check the adjacent entries for the next largest score

Move to the entry with the largest score and continue the path

Backtracking

Finds the best local alignment from the scoring matrix

Backtracking



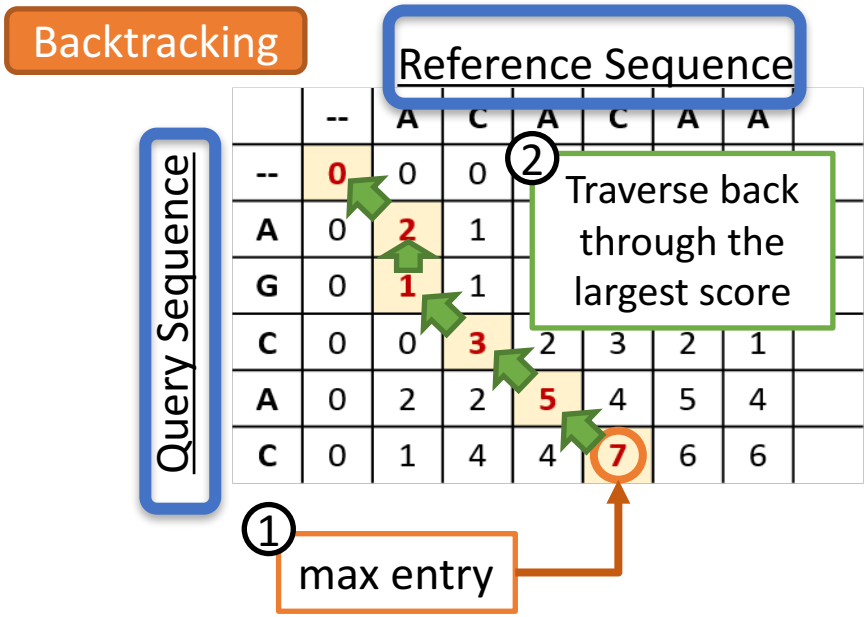
Step 3.

Get the resulting alignment

Path Direction	Alignment
Horizontal	Deletion
Vertical	Insertion
Diagonal	Match

Backtracking

Finds the best local alignment from the scoring matrix



	--	A	C	A	C	A	A
--	0	0	0				
A	0	2	1				
G	0	1	1				
C	0	0	3	2	3	2	1
A	0	2	2	5	4	5	4
C	0	1	4	4	7	6	6

Step 3.

Get the resulting alignment

Reference: A–CAC

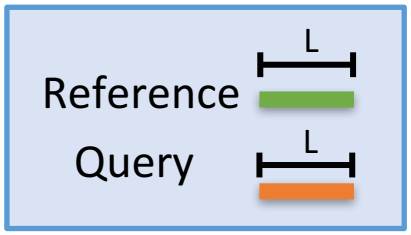
Query: AGCAC

Insertion

Alignment Score: 7

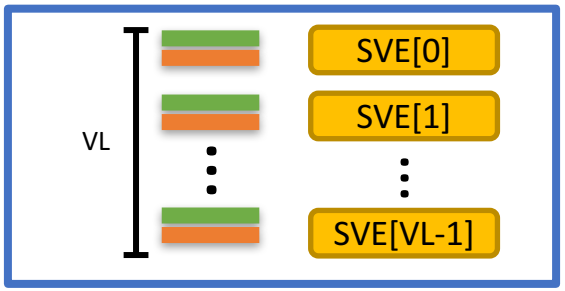
Smith-Waterman Vectorization

$$VL = \frac{\text{SIMD\_WIDTH}}{32 \text{ bits}}$$



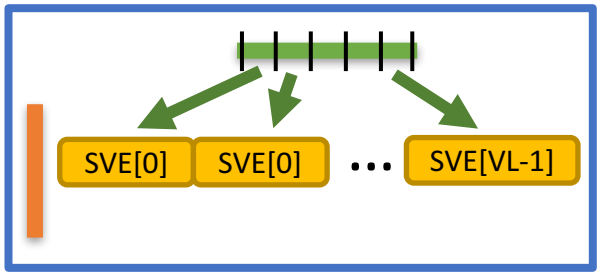
X 1000 samples

BATCH:



X (1000/VL) iterations

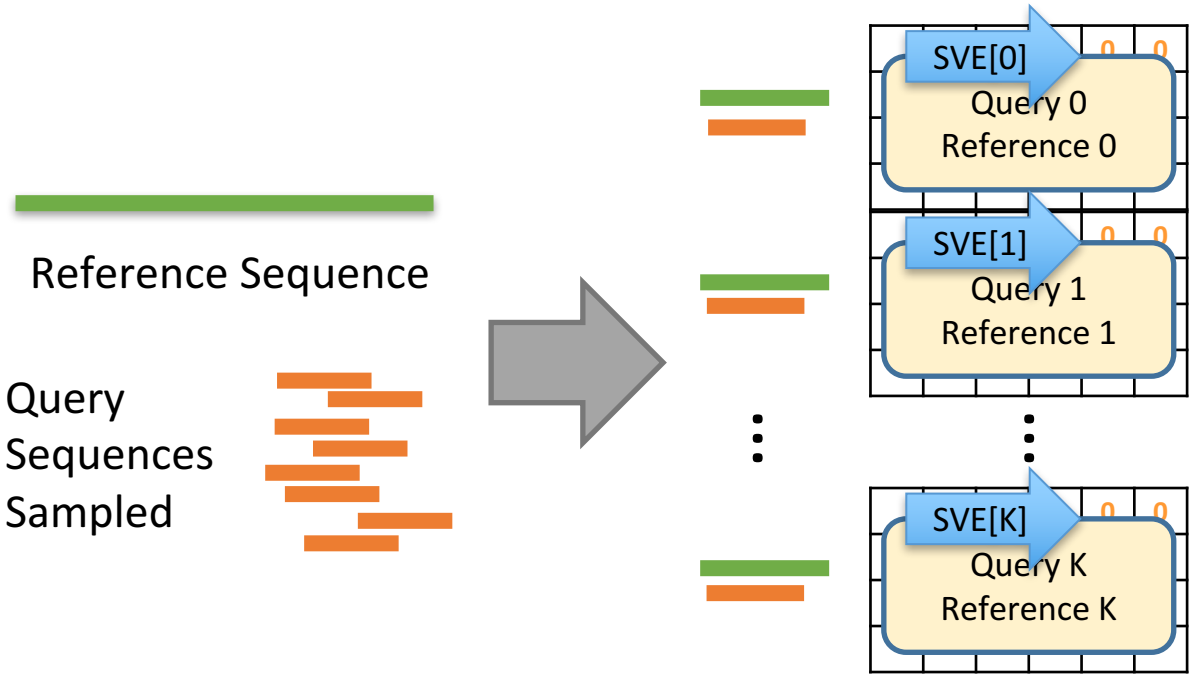
SLICED:



X 1000 iterations

Alignment Location & Score

Batch Smith-Waterman

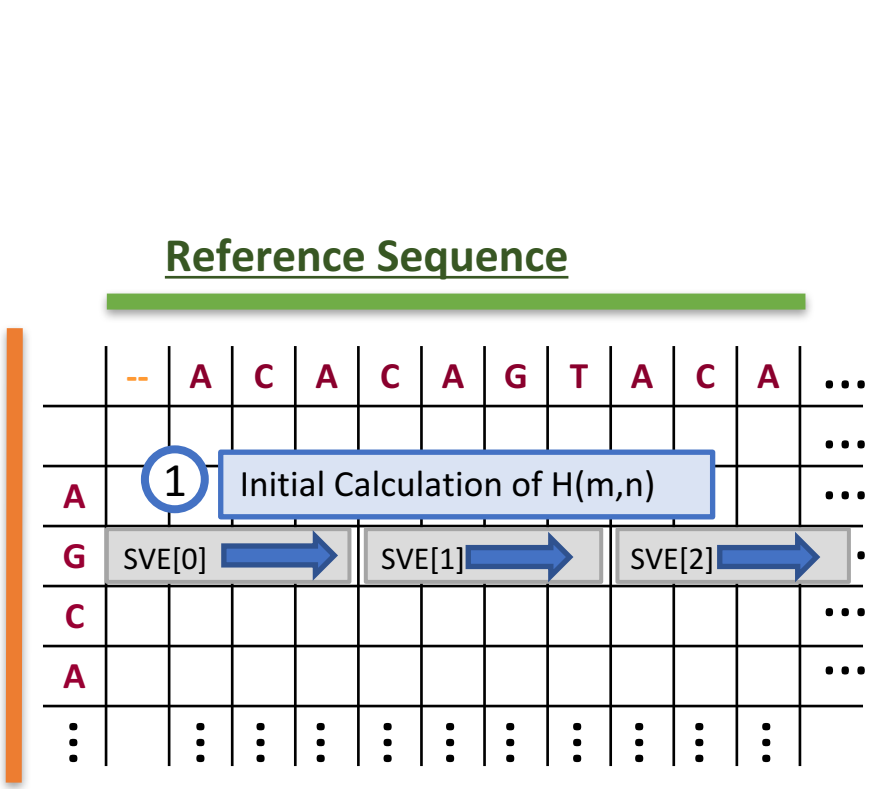


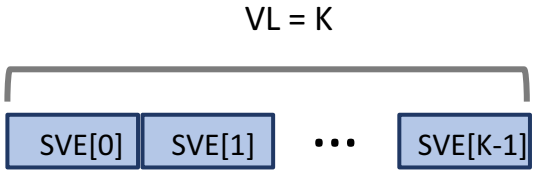
✓ No data dependencies between lanes

✘ Divergent memory access

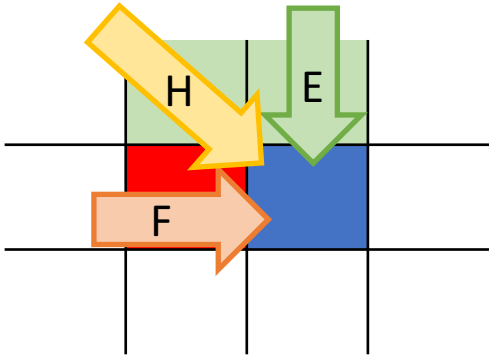
✘ More memory bandwidth needed

Sliced Smith-Waterman

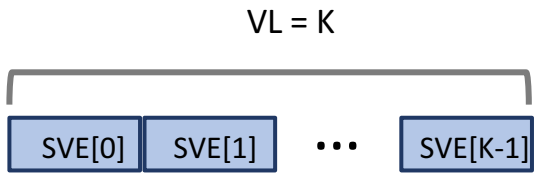


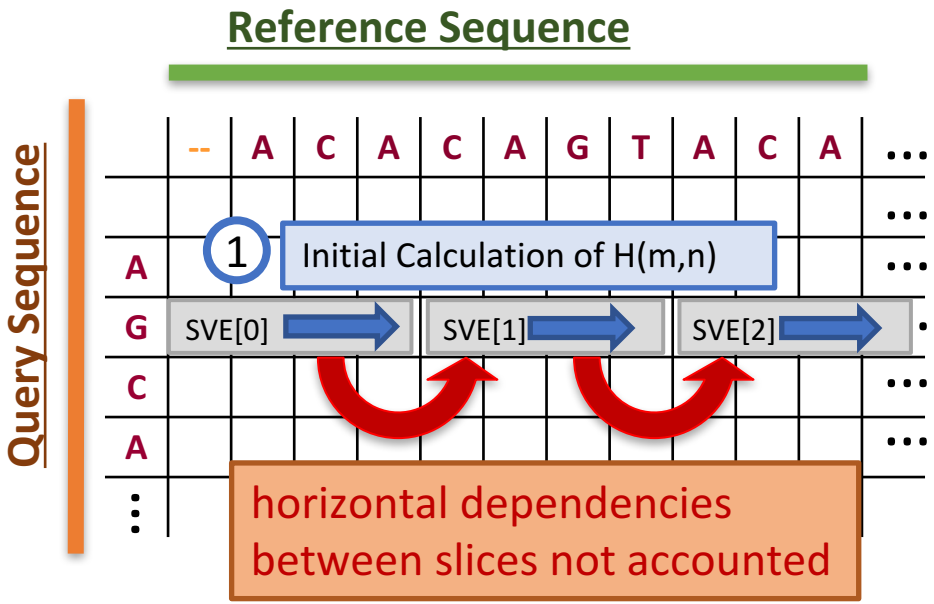


$$H(m,n) = max \begin{cases} E(m,n) \\ F(m,n) \\ H(m-1,n-1) + S(a_m, b_n) \end{cases}$$

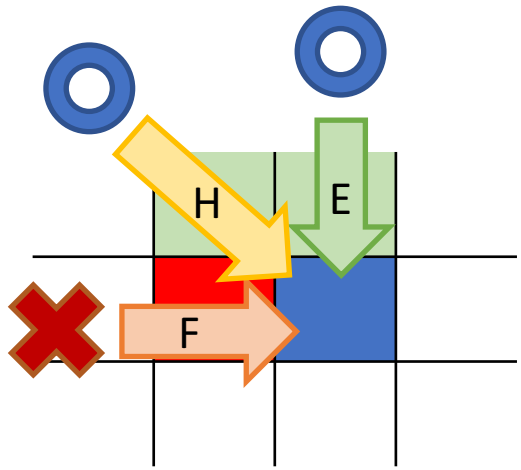


Sliced Smith-Waterman

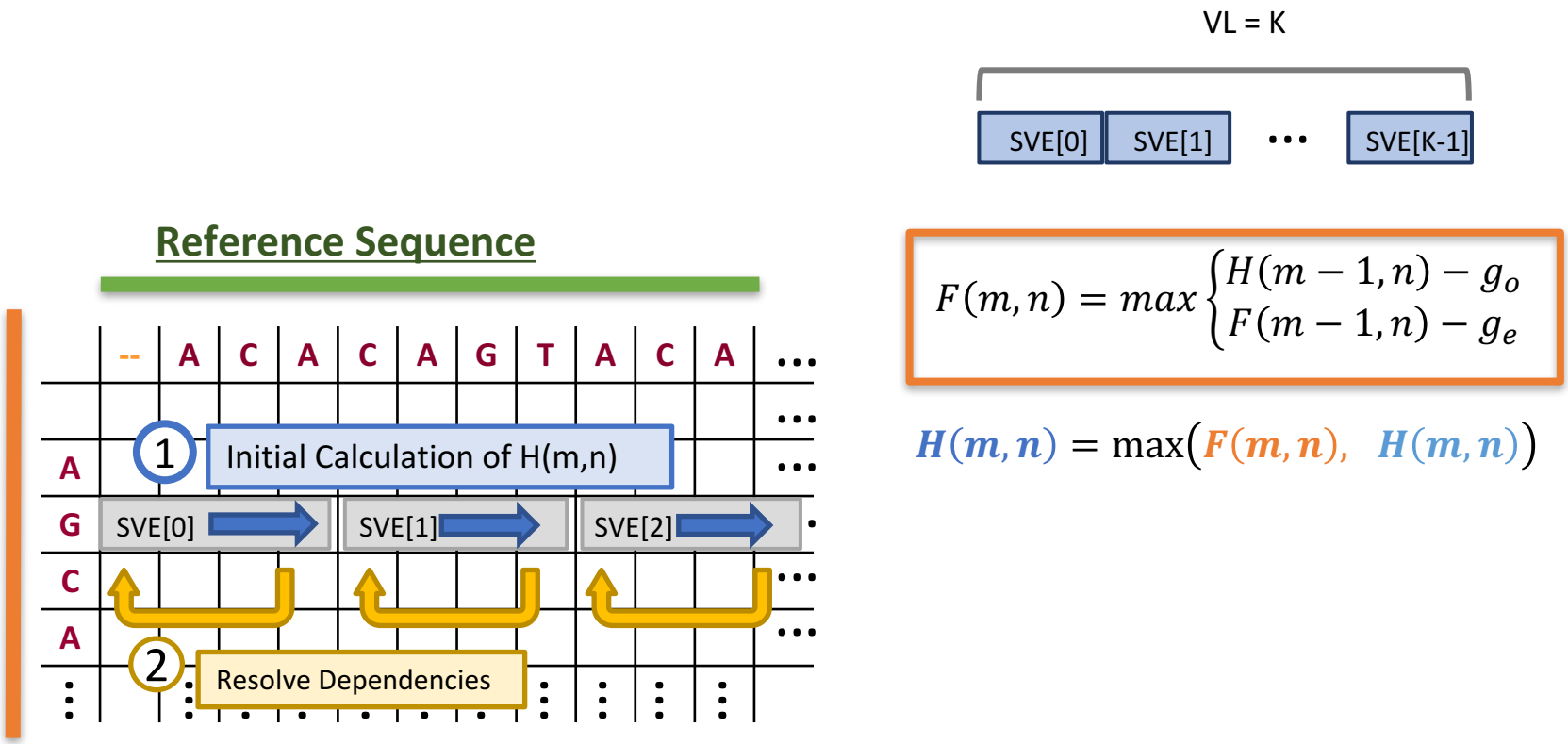




Value of F need to be re-calculated



Sliced Smith-Waterman

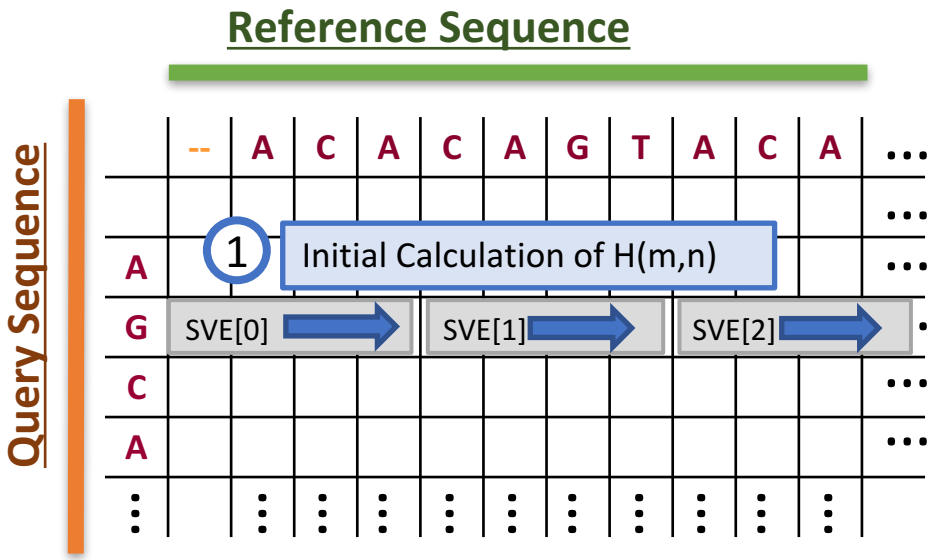


$$F(m,n) = max \begin{cases} H(m-1,n) - g_o \\ F(m-1,n) - g_e \end{cases}$$

$$H(m,n) = \max\left(F(m,n),\ H(m,n)\right)$$

Costly resolution as one need to traverse through the entire row

Sliced Smith-Waterman



✓ Smaller memory footprint

✓ Coalesced memory access

✘ Inter-lane dependencies

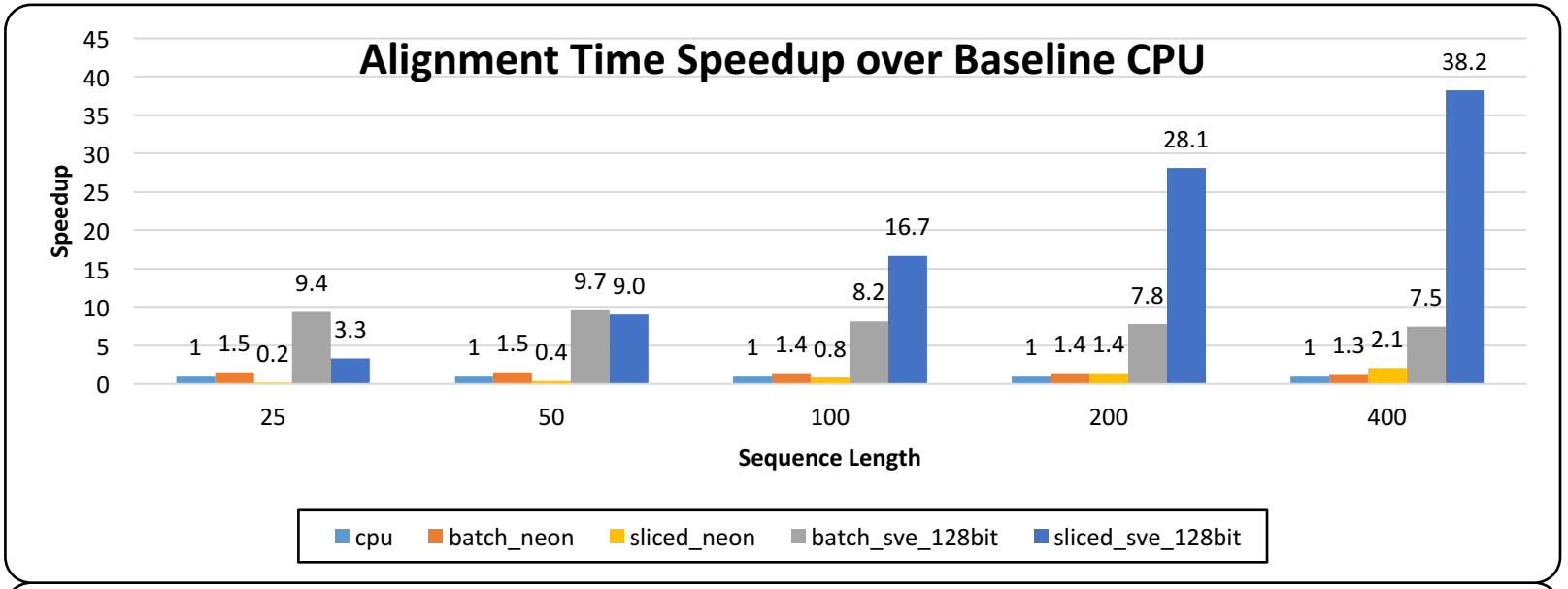
Experimental Setup

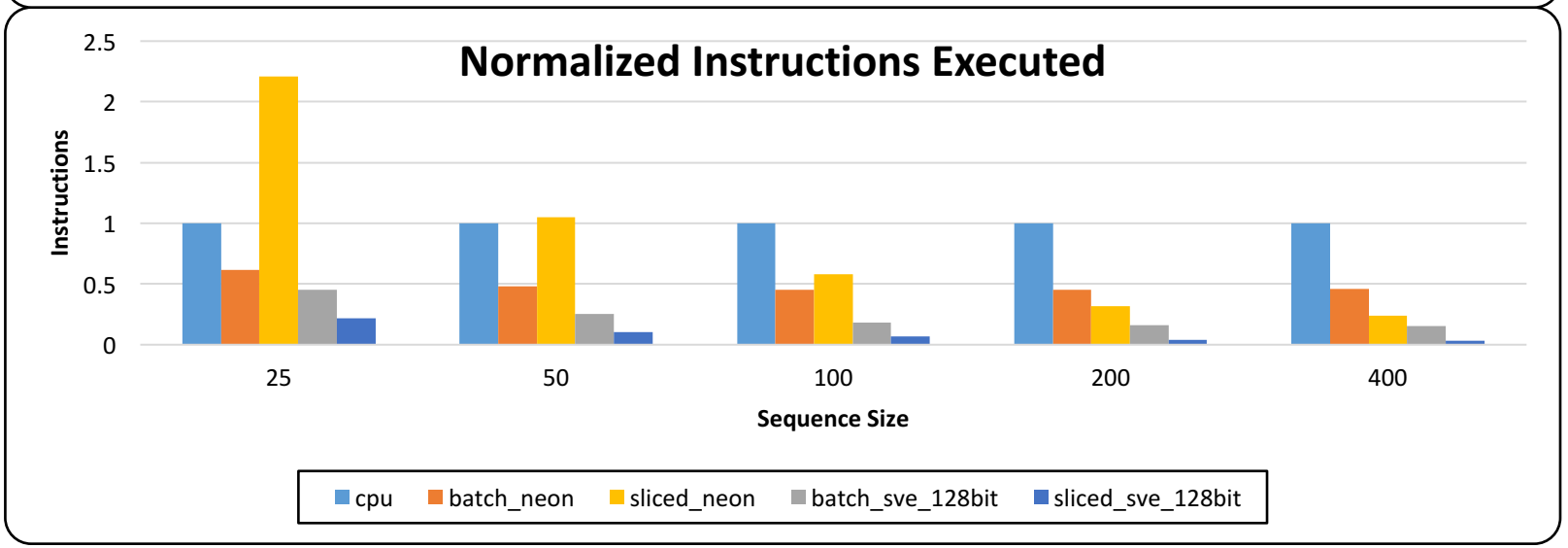
- Custom implementations run on modified gem5 setup supporting SVE instruction set

Component	Configuration
Core	Single-Core out-of-order 64-bit ARM, 1GHz, 8-issue SIMD Width: 128-bit (NEON), 128/256/512/1024-bit (SVE)
Cache	32KB private L1 instruction cache, 2-way associative 64KB private L1 data cache, 2-way associative 4MB private L2 inclusive cache, 8-way associative
DRAM	Capacity: 8GB Latency: 30 ns Memory Controller Bandwidth: 12.8 GB/s

- Current gem5 model serializes memory accesses across cache lines
- 25 to 400 sequence length

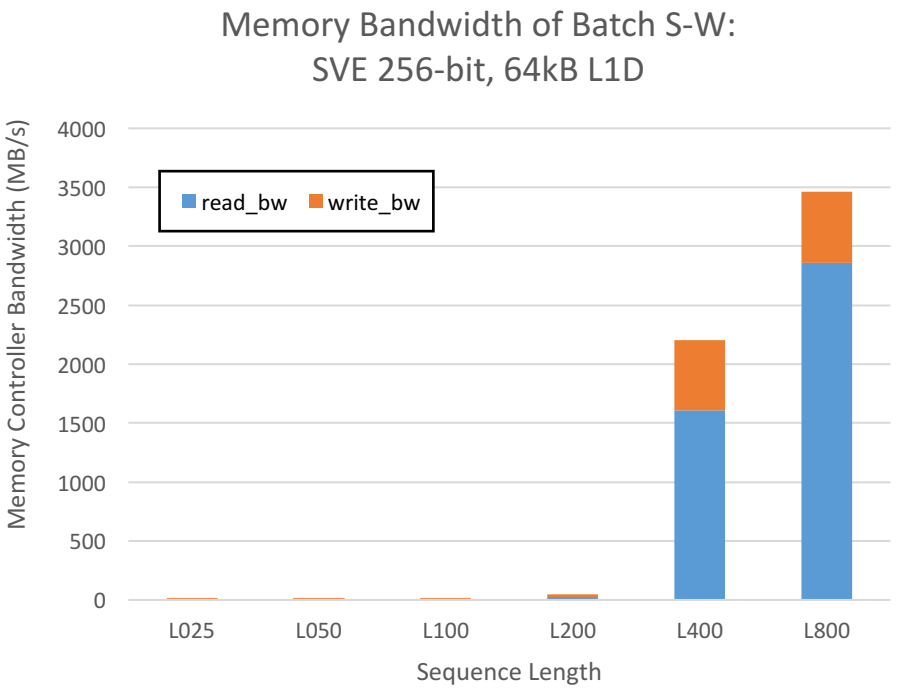
Performance

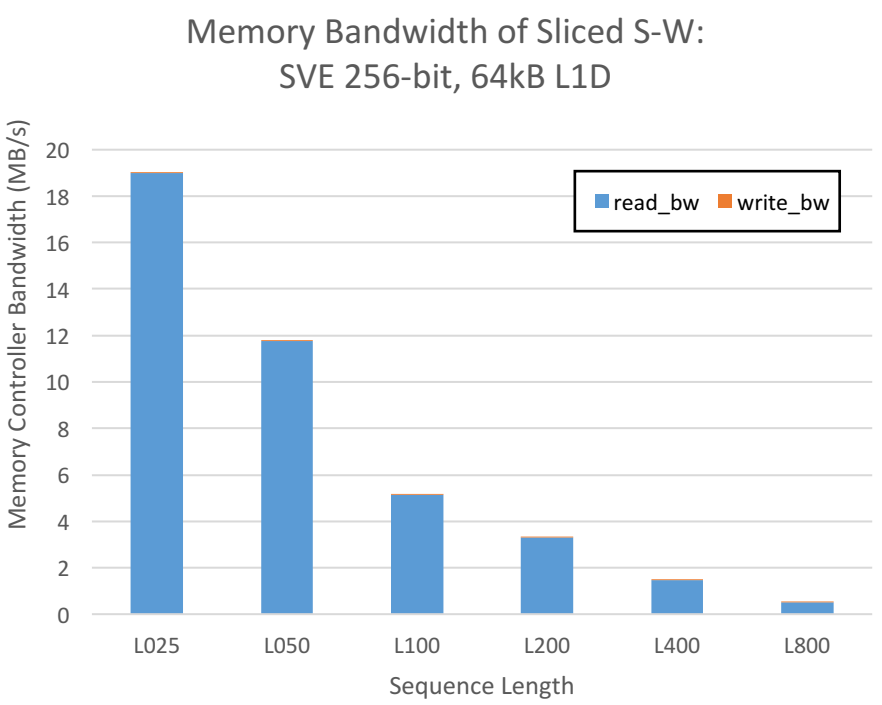




Batch vs Sliced: Memory Bandwidth

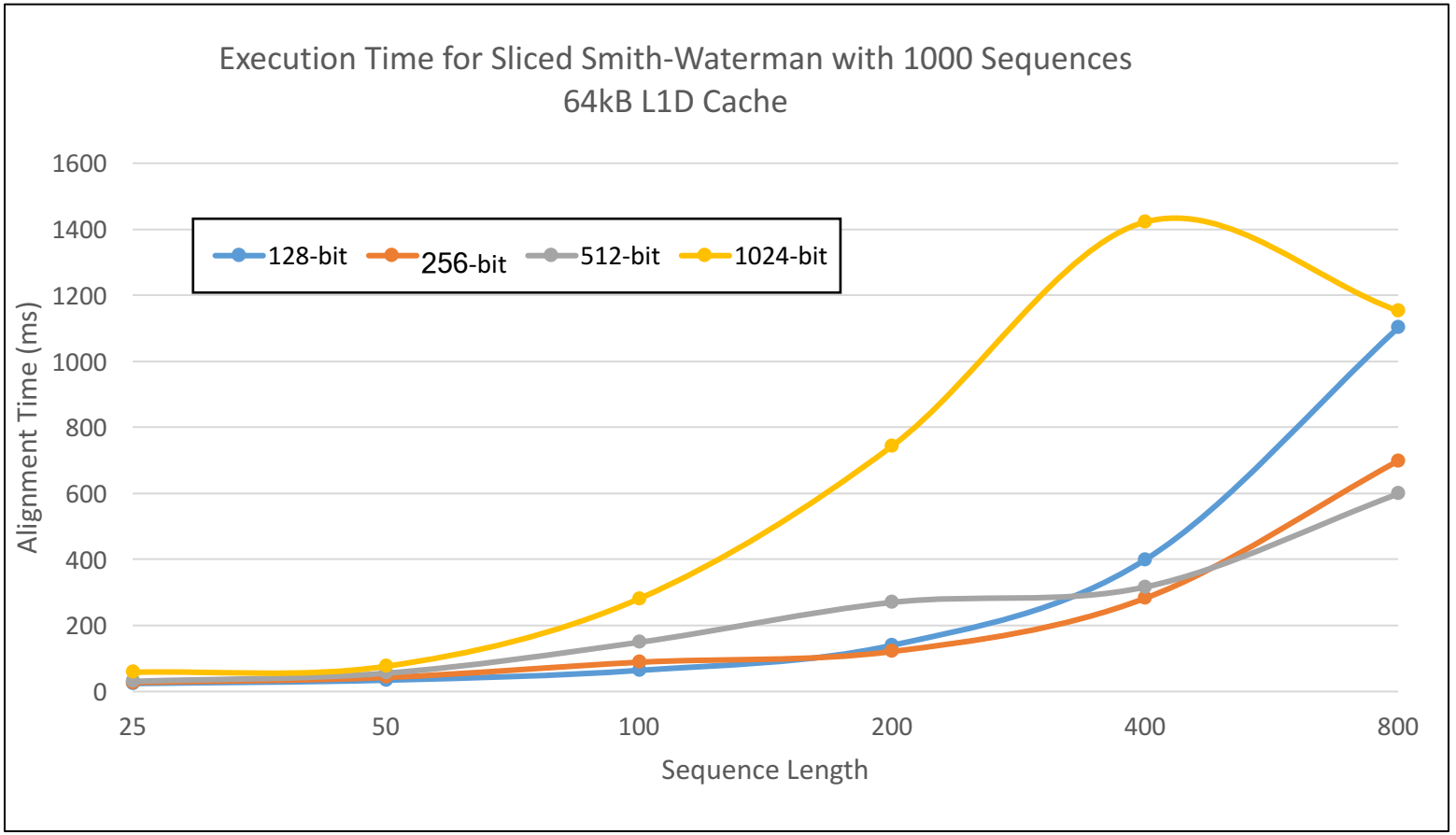
- Sliced reduces memory bandwidth





Sliced Smith-Waterman

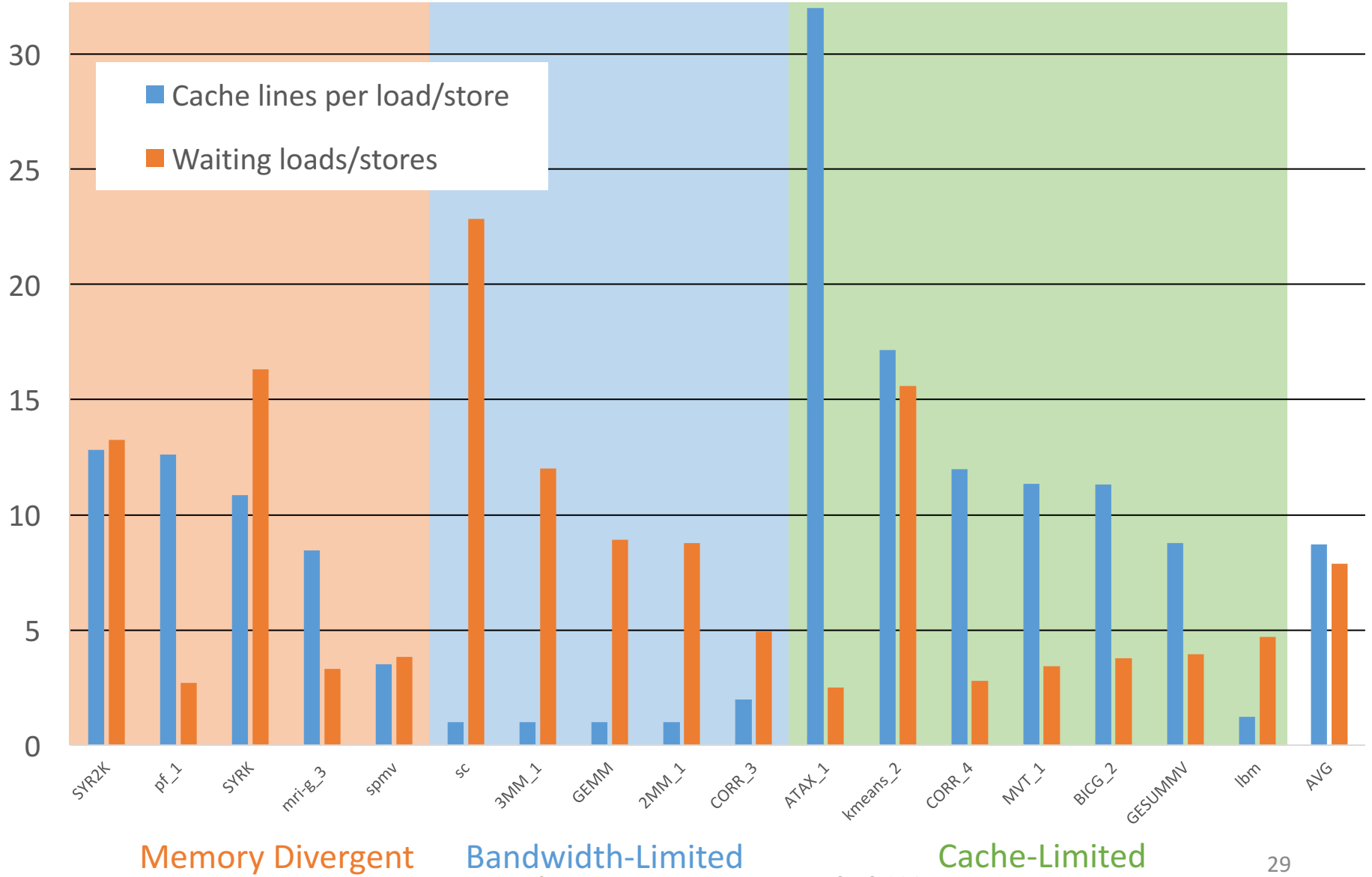
- Larger vector length only beneficial for longer sequences



Memory coalescing

- Short bursts of memory divergence is a problem in our current implementation
- While work is being done to implement scatter-gather functionality, we believe more can be done to exploit memory level parallelism
- We've demonstrated appreciable speedup on GPUs by coalescing not just across lanes in a vector, but across instructions as well
- We believe this can be exploited to a lesser degree on SIMD pipelines as well

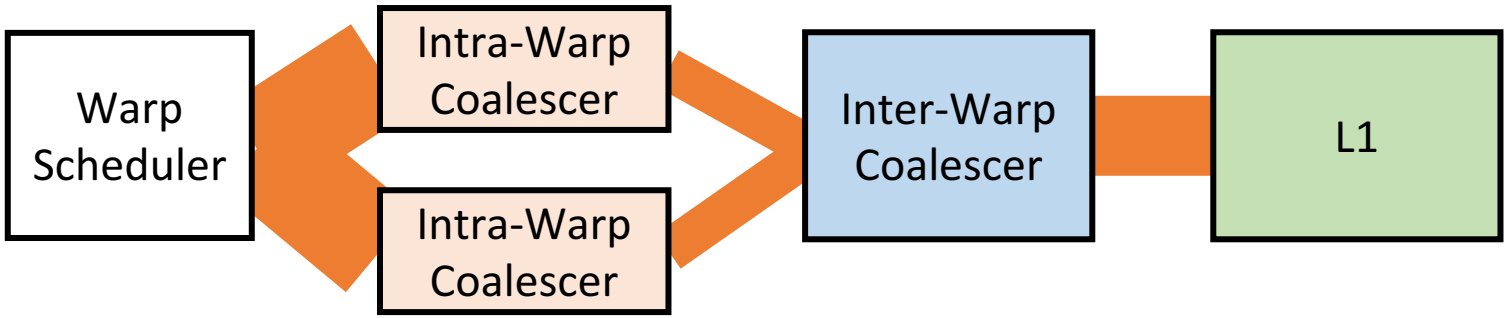
Hazards in Benchmarks

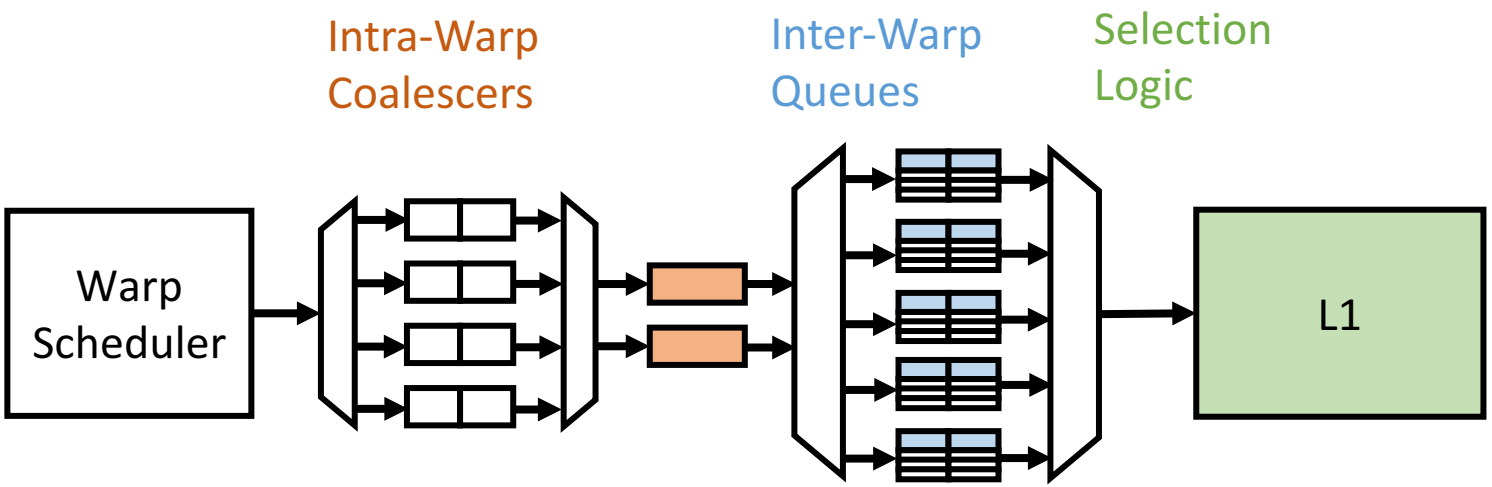


29

"WarpPool: sharing requests with inter-warp coalescing for throughput processors." *MICRO* 2015.

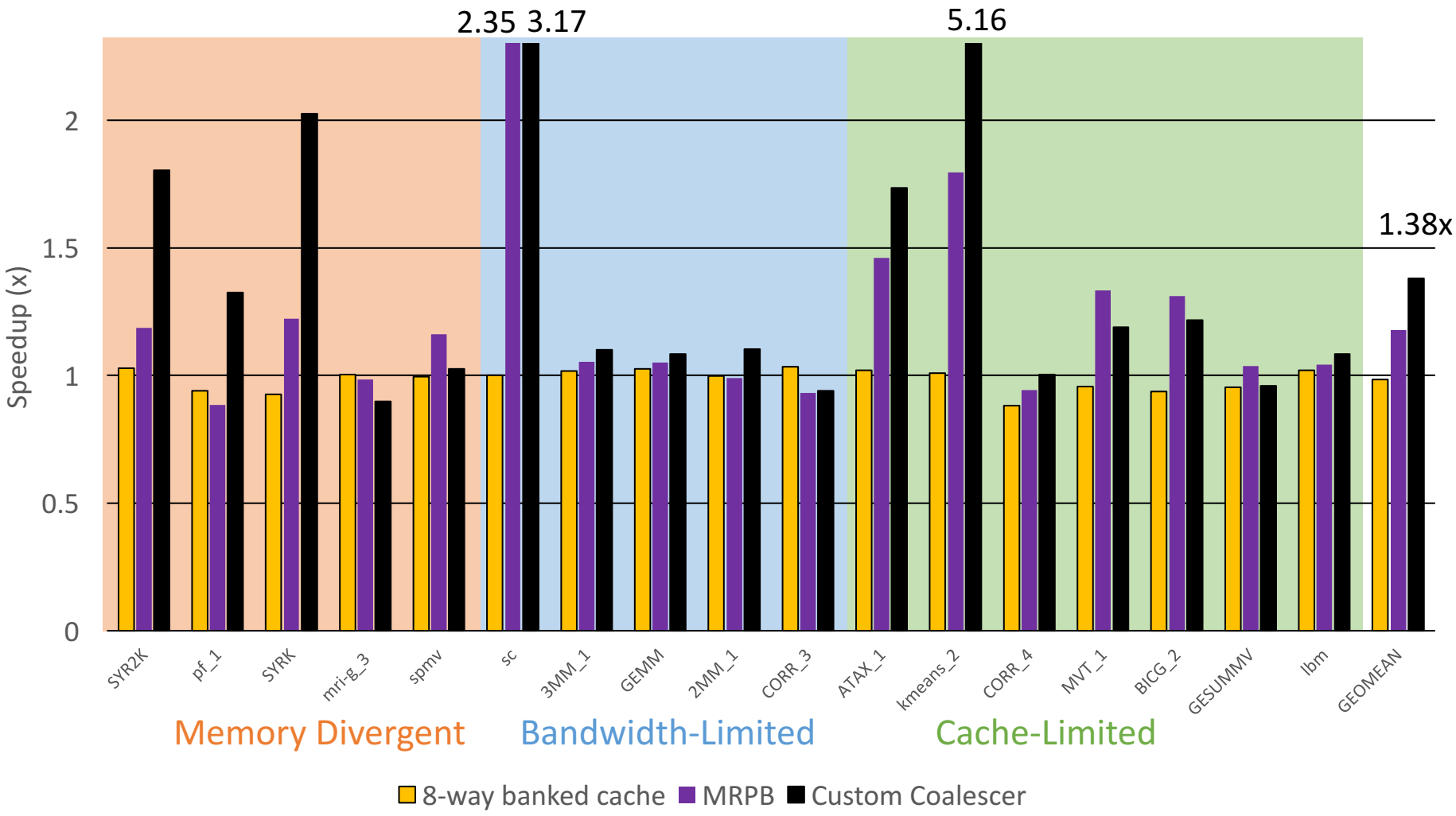
Memory Coalesing





"WarpPool: sharing requests with inter-warp coalescing for throughput processors." *MICRO* 2015.

Coalescing Speedup on GPUs



[1] MRPB: Memory request prioritization for massively parallel processors: HPCA 2014

Conclusion

- SVE instructions are well tailored to significantly reduce code size and execution overhead of HPC applications
- For fine-grained applications such as genomics, lack of scalable memory resources result in diminishing returns for increased vector size
- Future focus on improving memory coalescing may be the key to better performance

Questions?

Batch vs Sliced: Execution Time

- Batch performs better for short sequences, sliced performs better for long