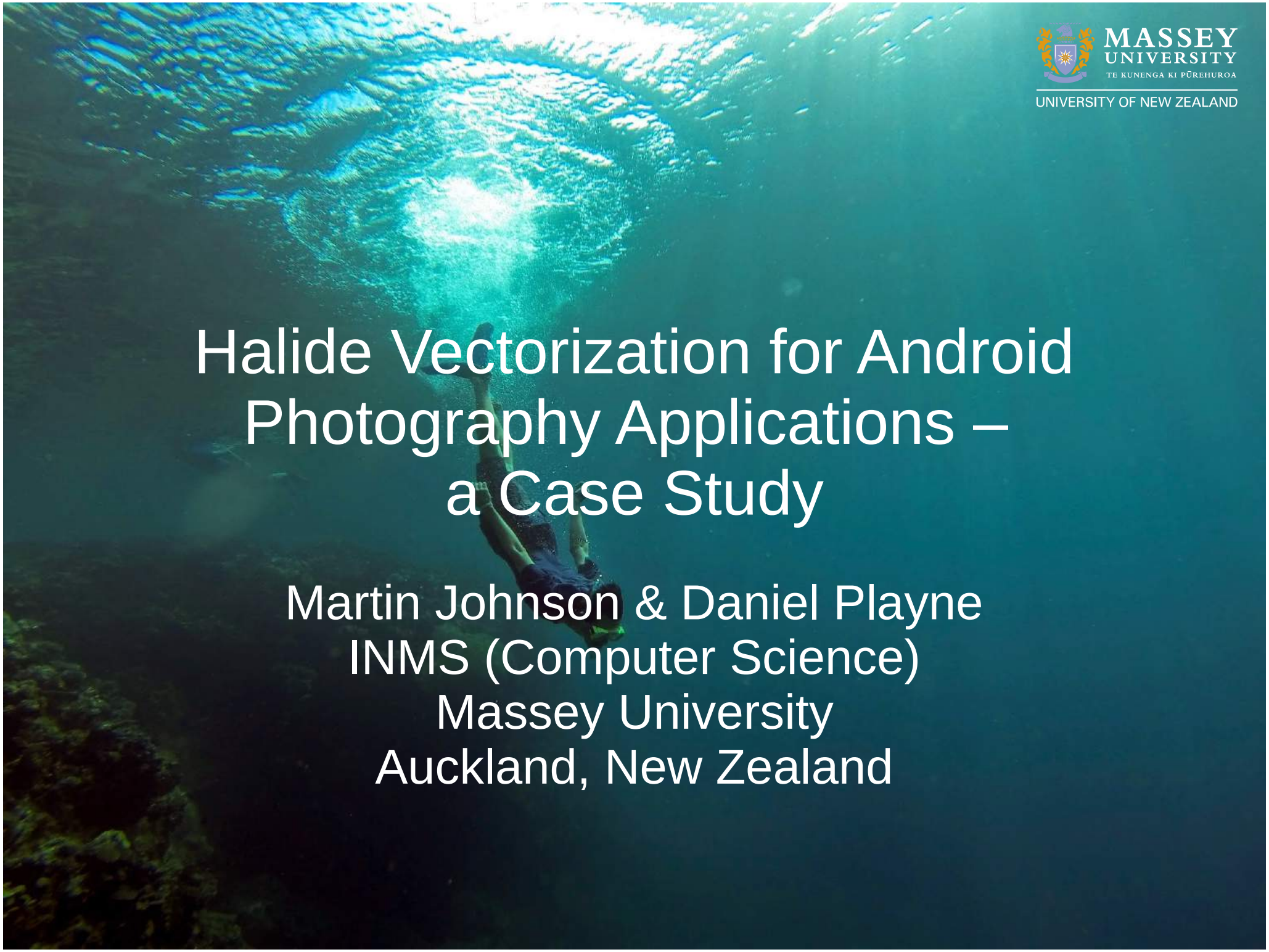


MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Halide Vectorization for Android Photography Applications – a Case Study

Martin Johnson & Daniel Playne
INMS (Computer Science)
Massey University
Auckland, New Zealand

Application Background

We have a successful HDR Camera App for Android mobile devices

- Top 10 in Paid Camera Apps on Google Play since 2013
- Computational Photography Native Library
  - CRF Estimation
  - Image Alignment
  - Frame Merging
  - Tonemapping
- Originally written in ARMv7 NEON Assembly Language
  - Hand optimized with vector instructions


12 MP Image


Same scene 33ms later


Image Alignment


Merged Image (6 frames)


Original Image Detail


Merged Image Detail

How to Vectorize?

C++ with Vectorizing Compiler
- Compilers are still bad at finding opportunities to vectorize imperative languages
- Side effects
- Bounds inference is difficult

Assembly Language
- Very Architecture dependent
- Time consuming

Compiler Intrinsics
- Architecture dependent
- Low Level

Halide
- Effective architecture independent vectorization

What is Halide?

A Domain Specific Language (DSL) for image processing

Developed by MIT from 2012

- http://halide-lang.org

Produces highly efficient parallel, vectorized code.

Claims to produce faster code than hand optimized assembly!

Uses LLVM with backends for:

- X86, ARMv7, ARMv8, CUDA, OpenGL, Hexagon
- Windows, OSX, Linux, Android, iOS

Parts of a Halide program

Variables – range is determined by bounds inference

```
Var x,y;
```

Expressions

```
Expr lum=77*red+150*green+29*blue;
```

Functions

```
Func sum_x;
sum_x(x,y)=sum(base(x * size + rTile, y))/size;
```

Reduction Domains – iterate over a specified range

```
RDom rTile(0, size);
```

Tuples – simple data structure indexed with []

```
Tuple t={xmin,ymin};
xoffset=t[0];
```

The algorithm is decoupled from its implementation

Each function can be scheduled separately

- Parallelize
- Vectorize
- Reorder
- Tile
- Unroll
- Split
- Compute_at
- Reorder

The schedule is guaranteed not to change the result

Allows you to easily exploit multiple cores, cache memory architecture and SIMD instructions

Halide Example - Image Histogram

```
RDom rx(0, input.width());
RDom ry(0, input.height());
Func row_hist,hist;

// the algorithm
row_hist(x, y) = 0;                    // create a histogram
row_hist(input(rx, y), y) += 1;        // for each row

hist(x) = sum(row_hist(x, ry));        // add them together

// the schedule
row_hist.compute_root().vectorize(x, 8).parallel(y);
row_hist.update().parallel(y);
hist.compute_root().vectorize(x, 8);
```

Halide Program for local alignment in the y direction

```
ImageParam base,other;          // input images
Func sumbase_x,sumother_x;      // tile row sums
RDom rTile(0, 32);              // domain for tile sum
RDom rExt(0,32);                // comparison extent
RDom rComp(-16, 32);            // comparison offset

// the alogrithm
sumbase_x(x, y) = sum(base(x*32+rTile,y))/32;
sumother_x(x, y) = sum(other(x*32+rTile,y))/32;
Expr sim_x = absd(sumbase_x(x, y*32+rExty),
                  sumother_x(x,y*32+rComp+rExty));
Tuple min_y = argmin3(rComp,sum(rExty,sim_x));
```

Vectorize

vectorize(N,TailStrategy)

N is the vector size, can be larger than the natural size.
What if the Image width is not a multiple of N?
Make sure image is padded at end
Could compute some pixels twice
Only a problem if input image is reused for output.
Use TailStrategy::GuardWithIf in this case

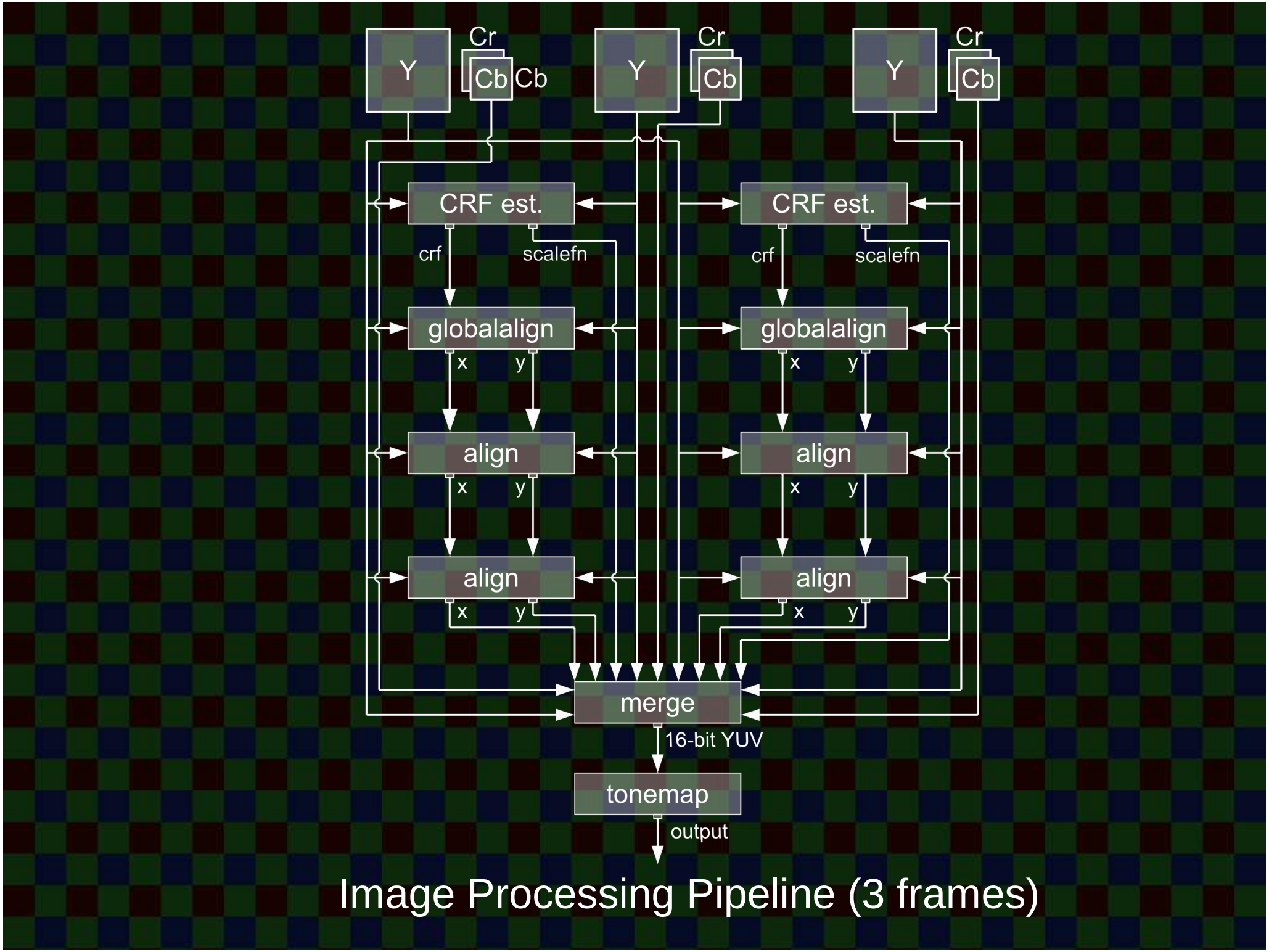


Image Processing Pipeline (3 frames)

Performance Measurement

Each function can be profiled separately, e.g.

```
total time: 56.326656ms samples: 849 runs: 6 time/run: 9.387776 ms
average threads used: 6.621908
heap allocations: 96 peak heap usage: 144368 bytes
 overhead:                0.076ms   (0%)
 sumbase_y:               0.131ms   (1%)
 sum$1:                   0.179ms   (1%)
 sumother_y:              0.215ms   (2%)
 sum$3:                   1.454ms   (15%)
 f:                       3.562ms   (37%)
 sum$4:                   0.211ms   (2%)
 sumbase_x:               0.671ms   (7%)
 sum:                     0.493ms   (5%)
 sumother_x:              0.771ms   (8%)
 sum$2:                   1.183ms   (12%)
 f$1:                     0.160ms   (1%)
 sum$5:                   0.277ms   (2%)
 f0:                      0.000ms   (0%)
```

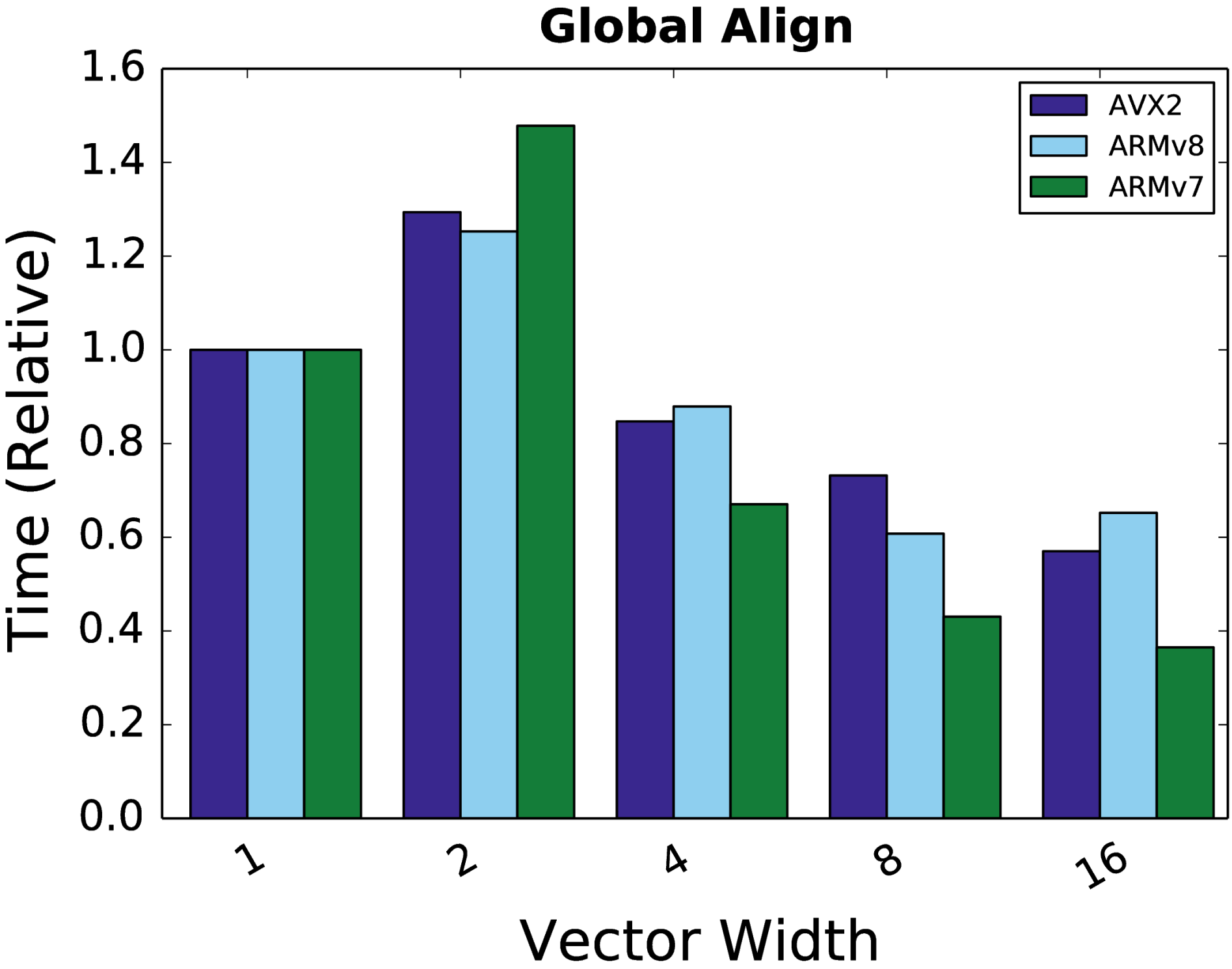
Performance Comparison

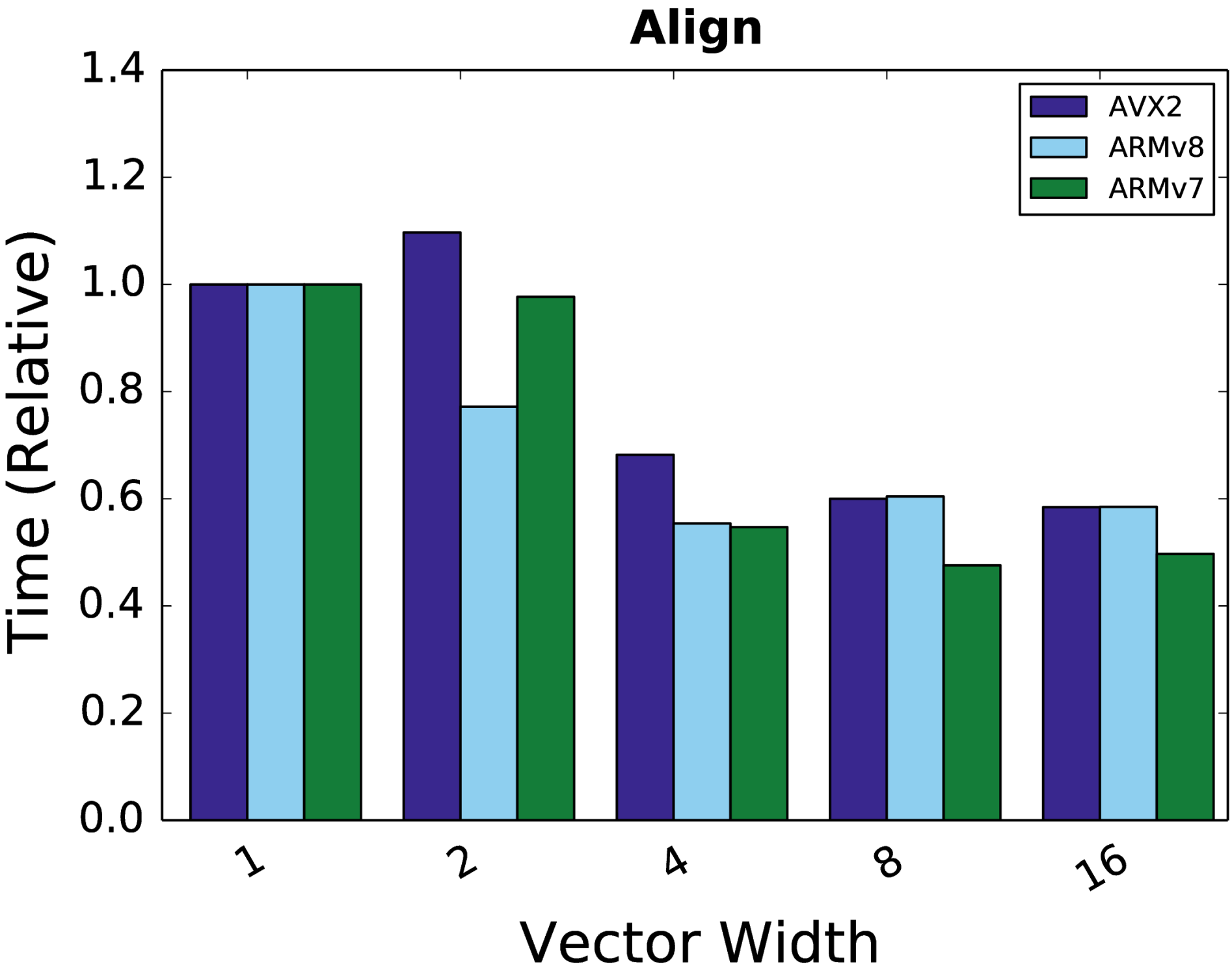
Stage	ARMv7 C++	ARMv7 NEON	ARMv7 Halide	ARMv8 Halide	AXV2 Halide
global align	81ms	48ms	27ms	34ms	9ms
align	-	-	74ms	62ms	13ms
merge	354ms	307ms	237ms	172ms	34ms
Tonemap	-	-	870ms	256ms	26ms

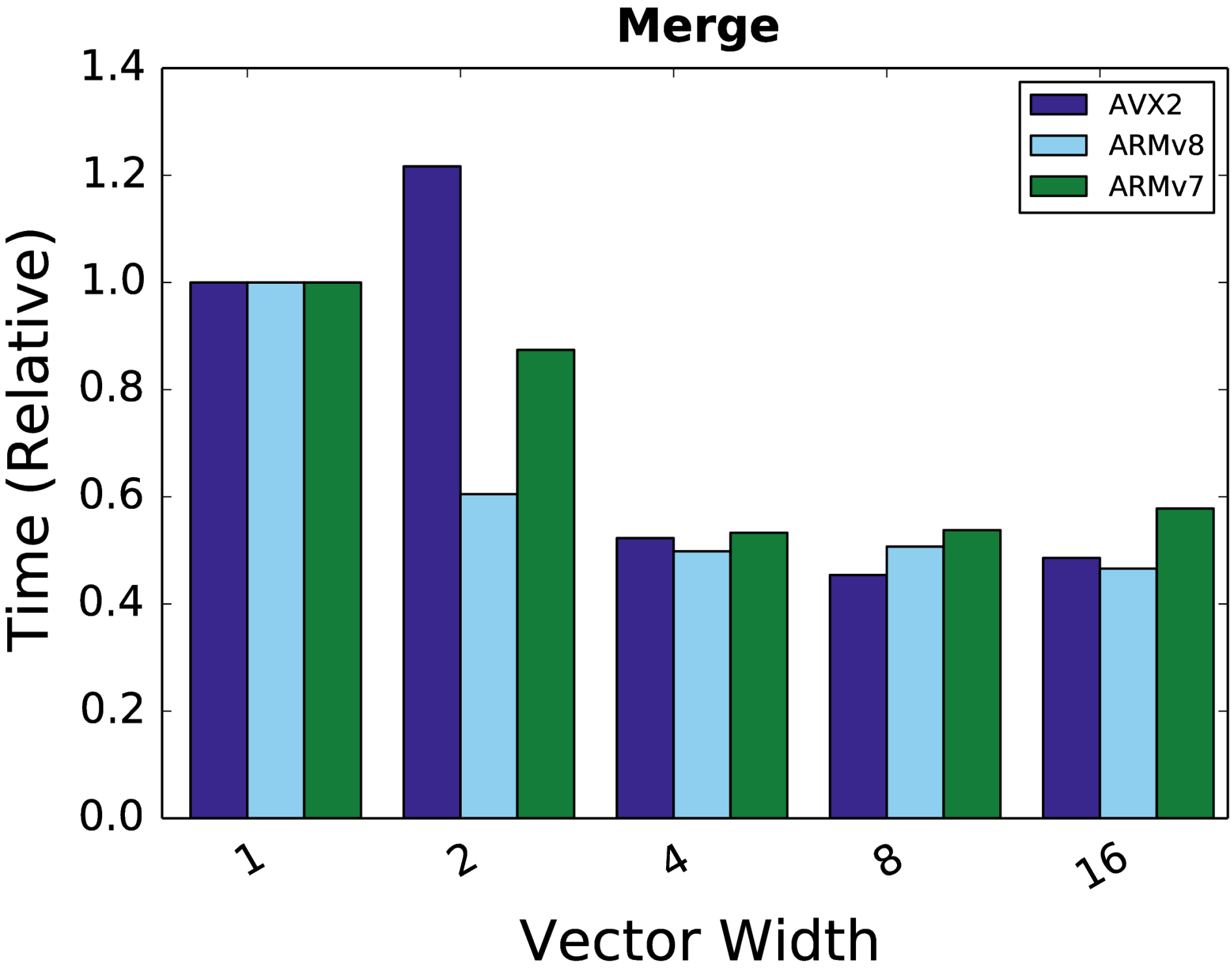
Pipeline Stage Results

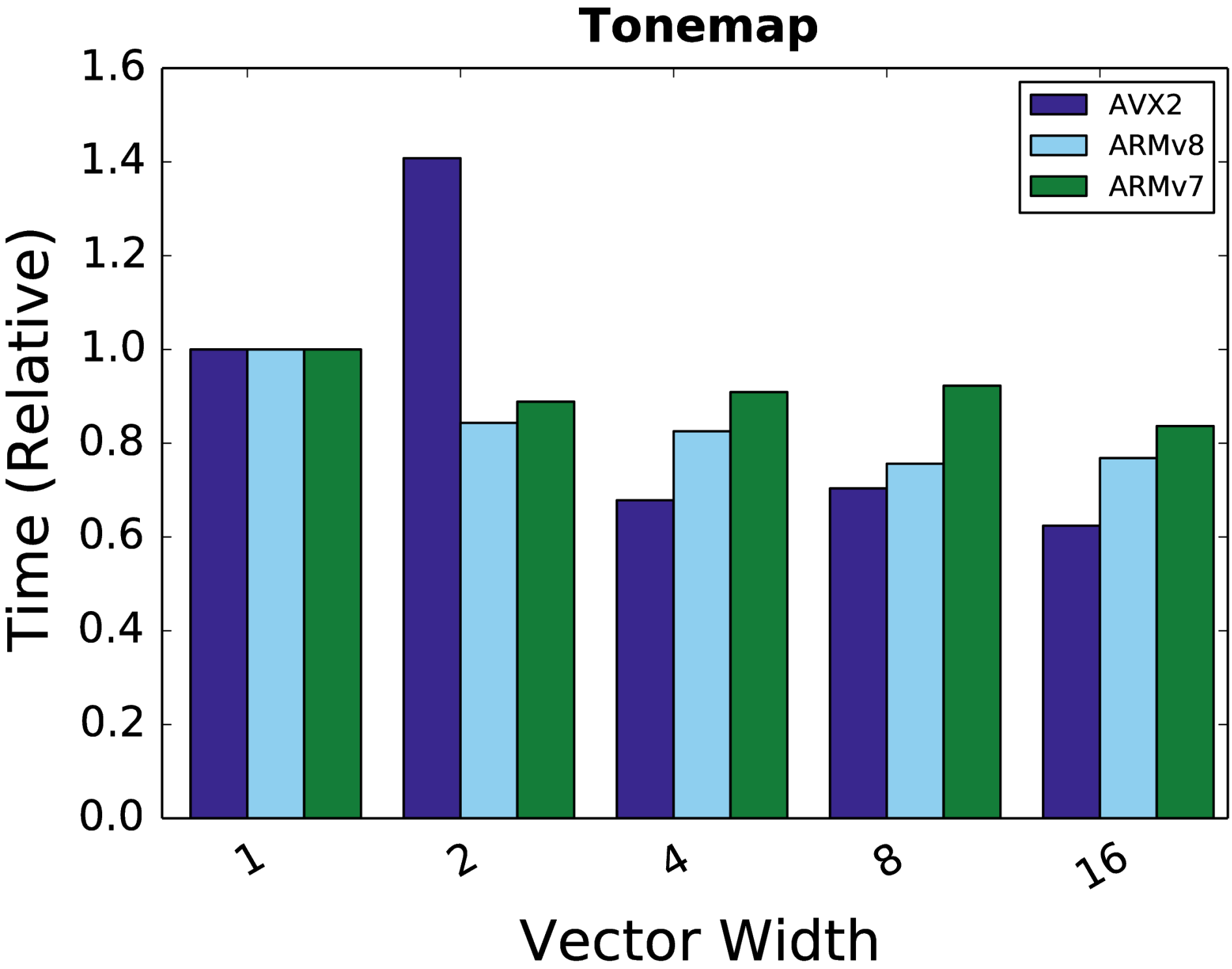
Each pipeline stage profiled on 3 architectures

- ARMV7 (32 bit with 128 bit vectors)
- ARMV8 (64 bit with 128 bit vectors)
- Intel AVX2 (64 bit with 256 bit vectors)


Global Align
Time (Relative)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
AVX2
ARMv8
ARMv7
1
2
4
8
16
Vector Width


Align
AVX2
ARMv8
ARMv7
Time (Relative)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
8
16
Vector Width

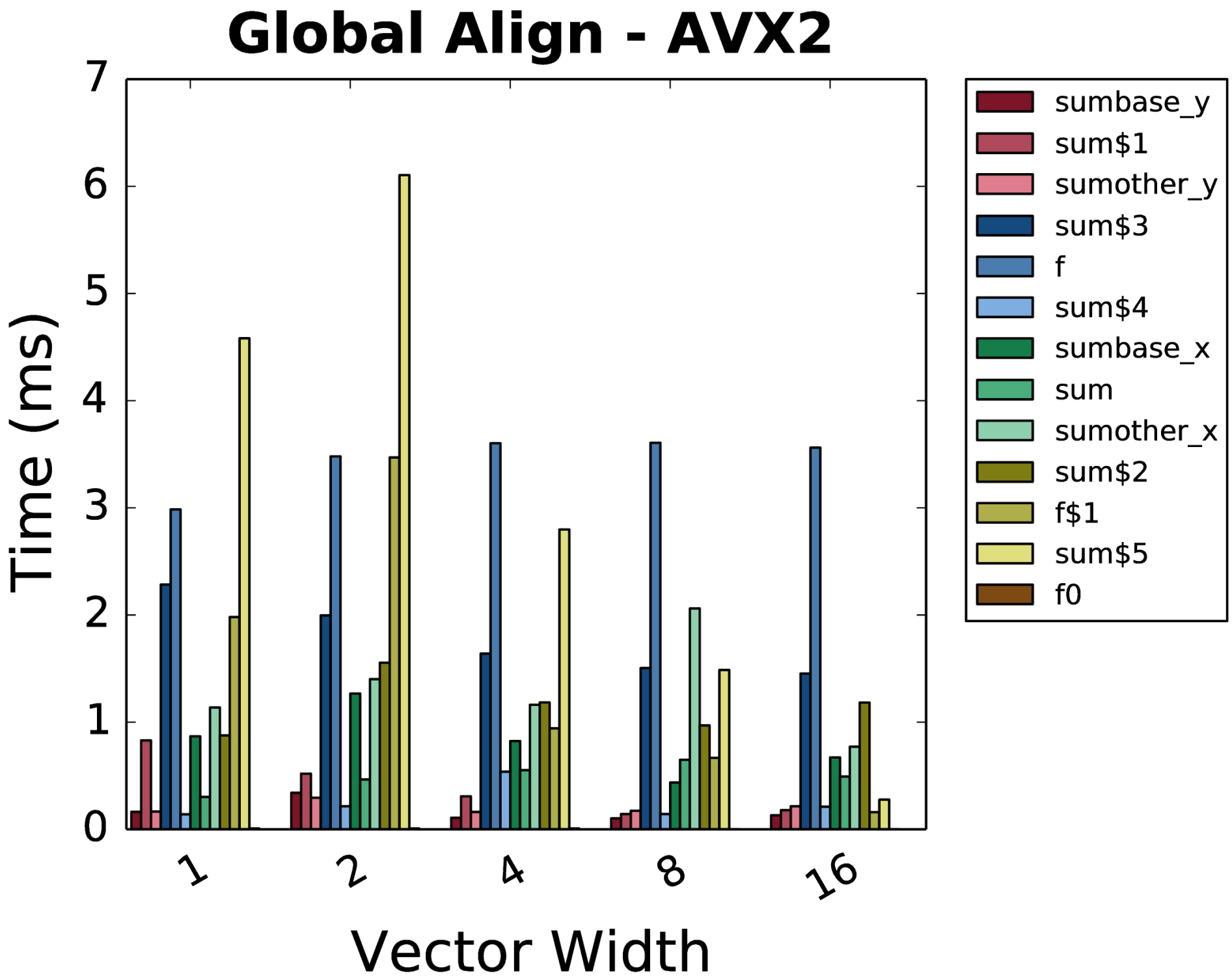

Merge
AVX2
ARMv8
ARMv7
Time (Relative)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
8
16
Vector Width

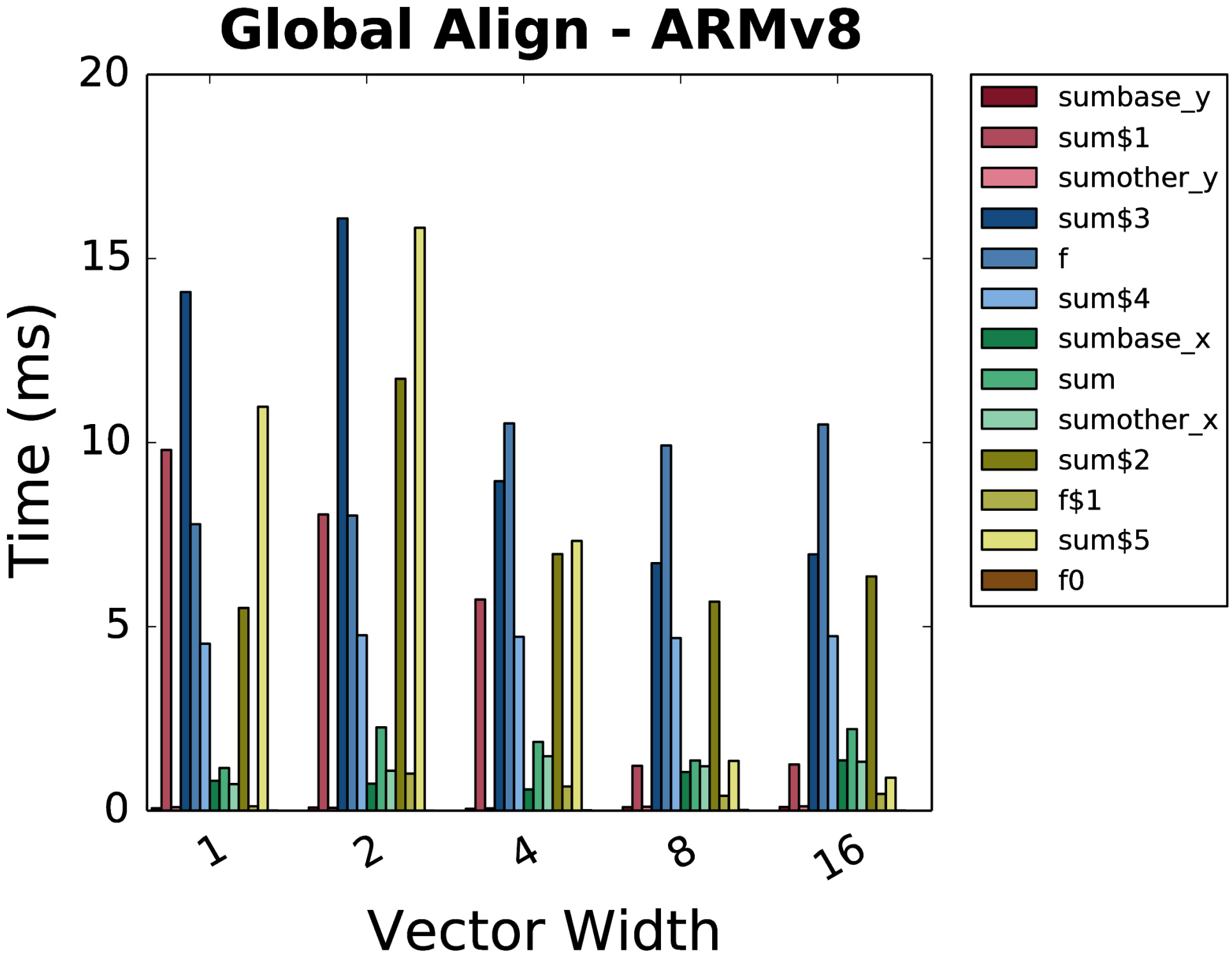

Tonemap
Time (Relative)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
AVX2
ARMv8
ARMv7
1
2
4
8
16
Vector Width

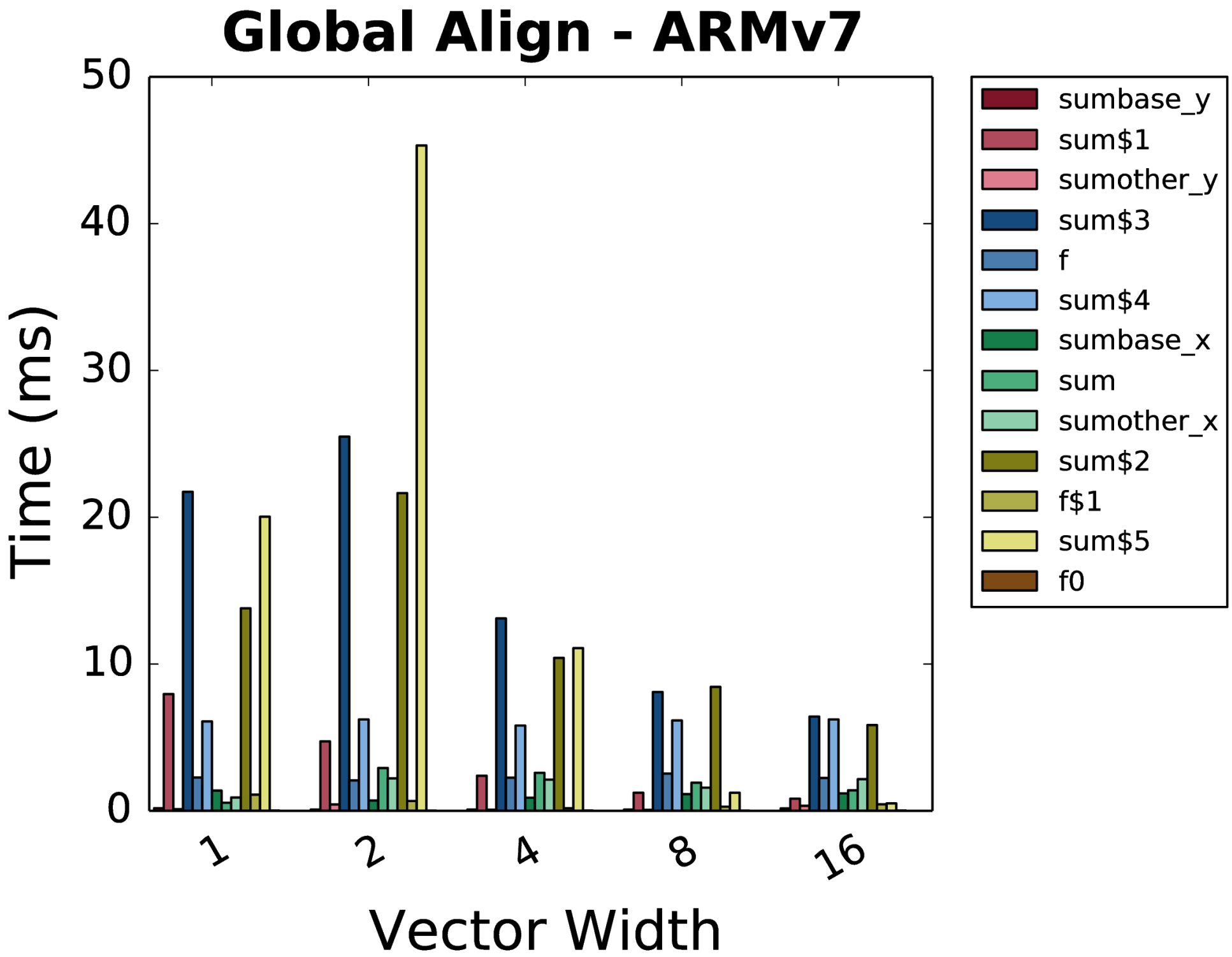
Fine Grained Results

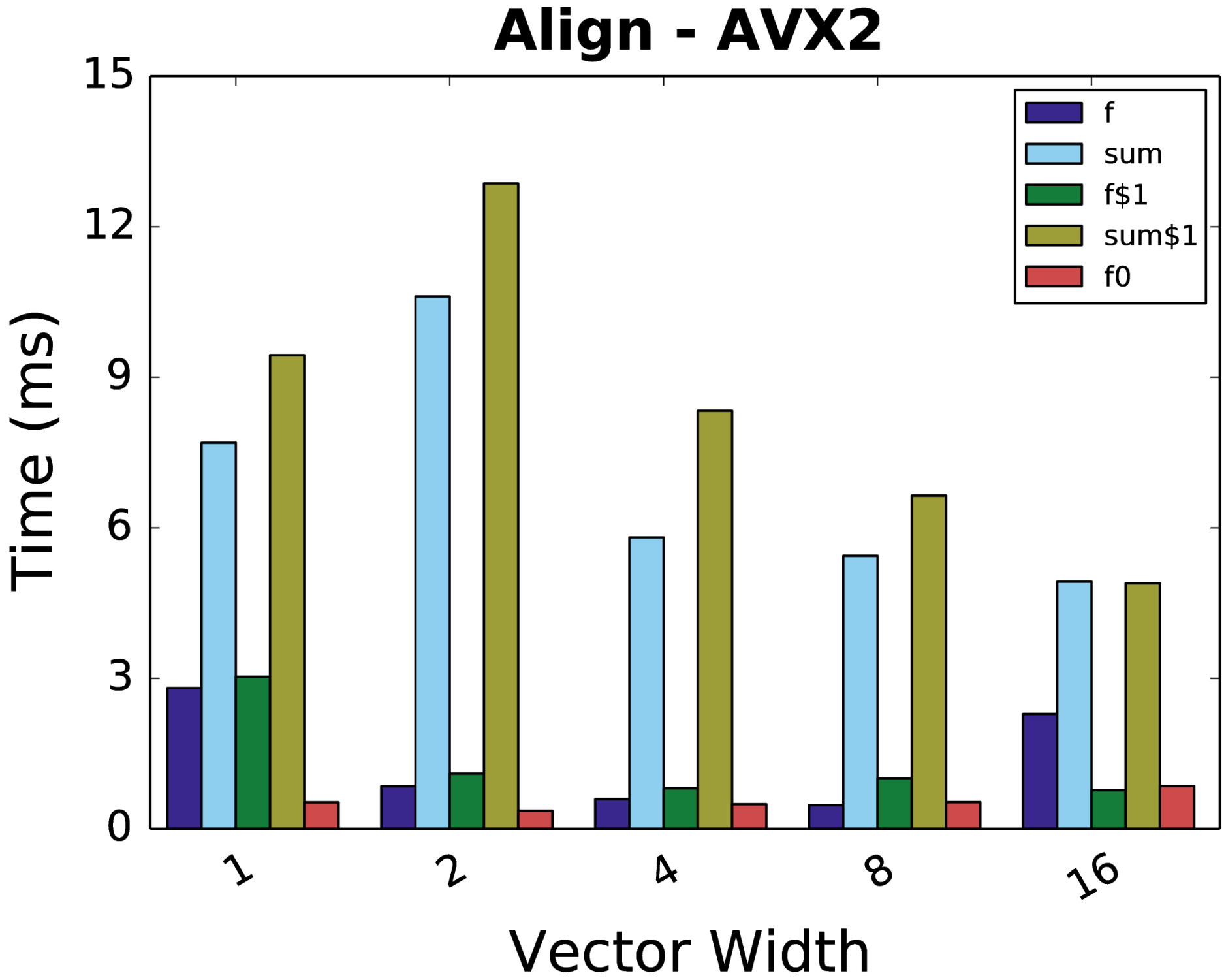
Each Halide function in the pipeline can be profiled separately.

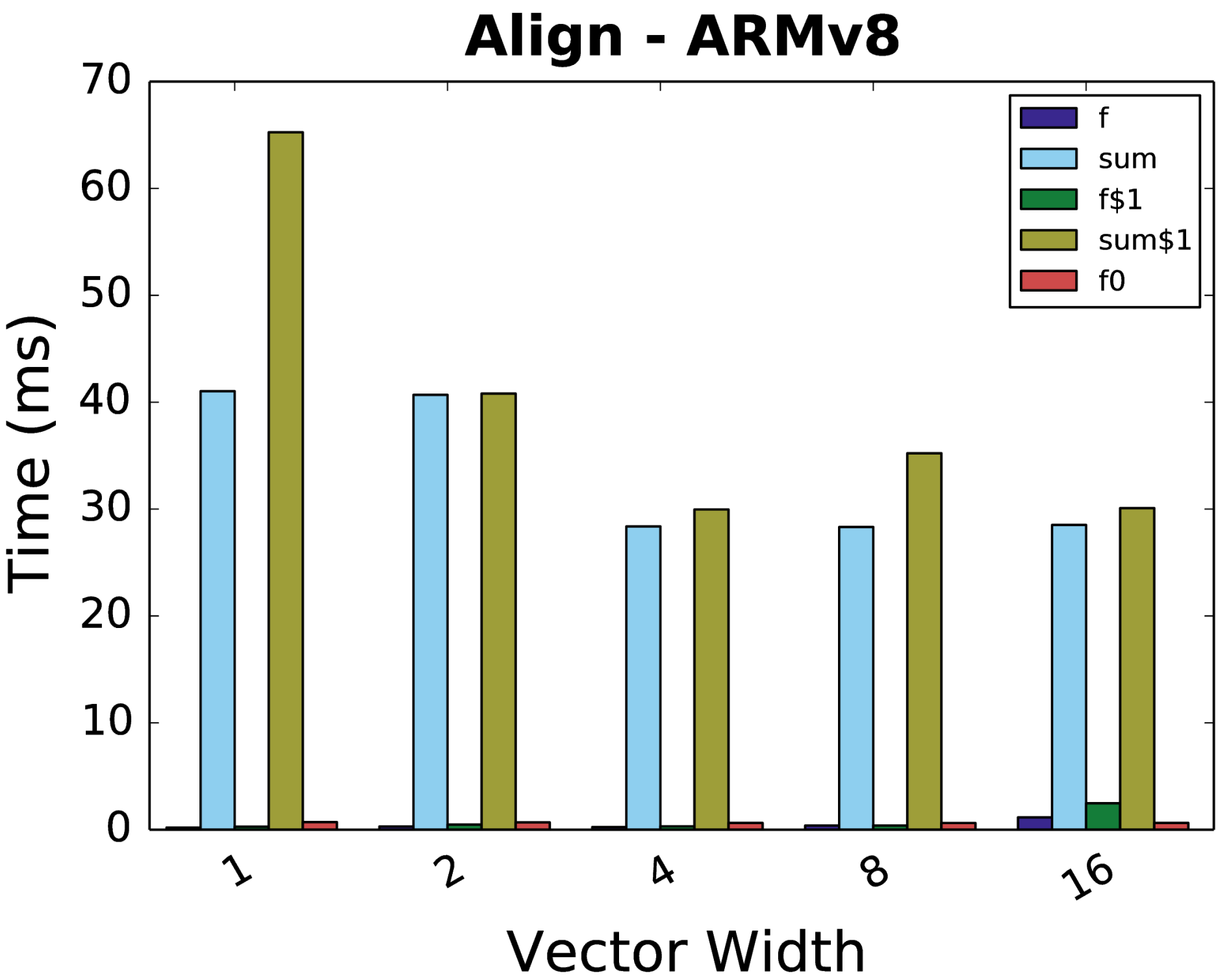
Global Align:	13 functions
Align:	5 functions
Merge:	2 functions
Tonemap:	5 functions

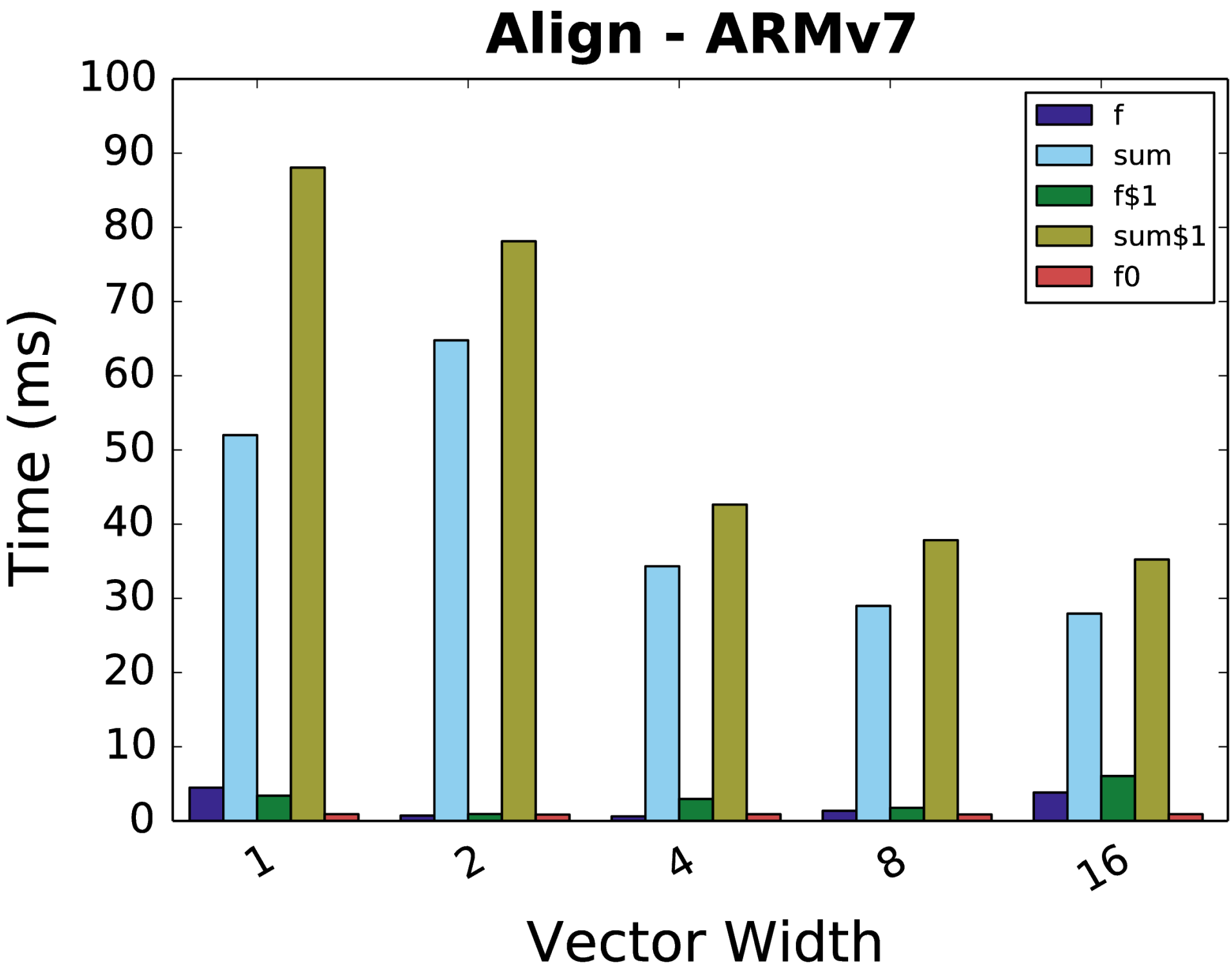

Global Align - AVX2
Time (ms)
0
1
2
3
4
5
6
7
1
2
4
8
16
Vector Width
sumbase_y
sum$1
sumother_y
sum$3
f
sum$4
sumbase_x
sum
sumother_x
sum$2
f$1
sum$5
f0

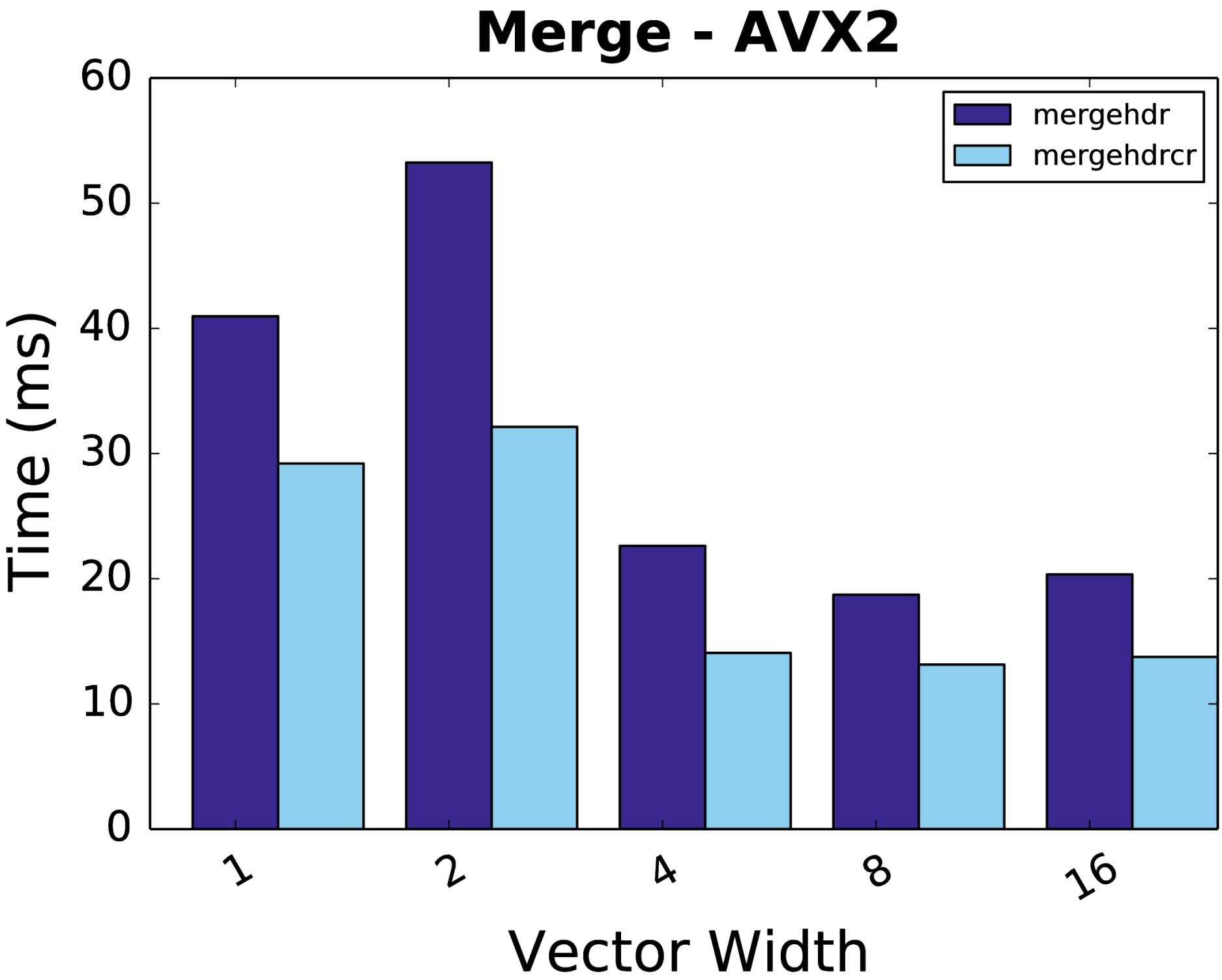

Global Align - ARMv8
Time (ms)
0
5
10
15
20
1
2
4
8
16
Vector Width
sumbase_y
sum$1
sumother_y
sum$3
f
sum$4
sumbase_x
sum
sumother_x
sum$2
f$1
sum$5
f0

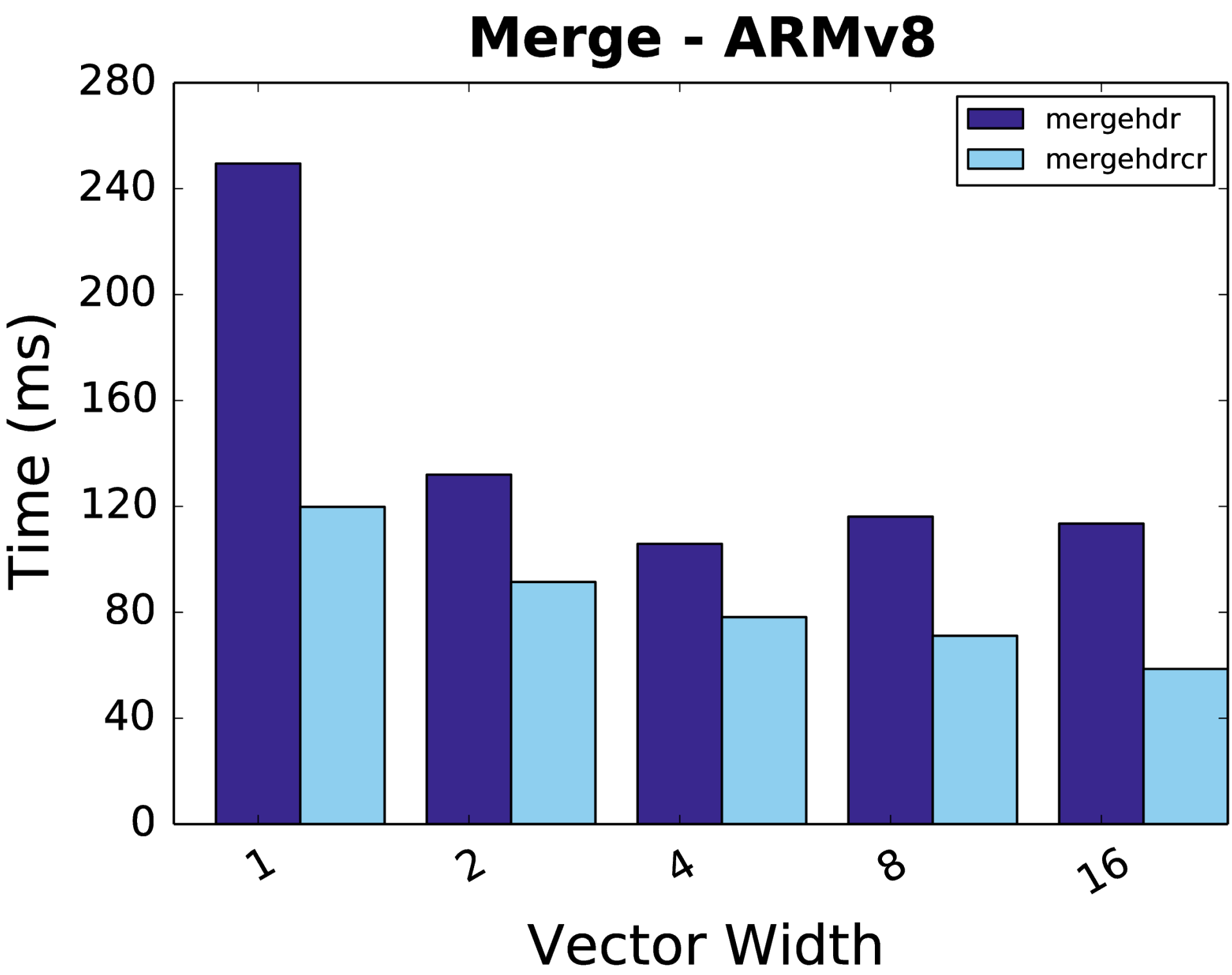

Global Align - ARMv7
Time (ms)
0
10
20
30
40
50
1
2
4
8
16
Vector Width
sumbase_y
sum$1
sumother_y
sum$3
f
sum$4
sumbase_x
sum
sumother_x
sum$2
f$1
sum$5
f0

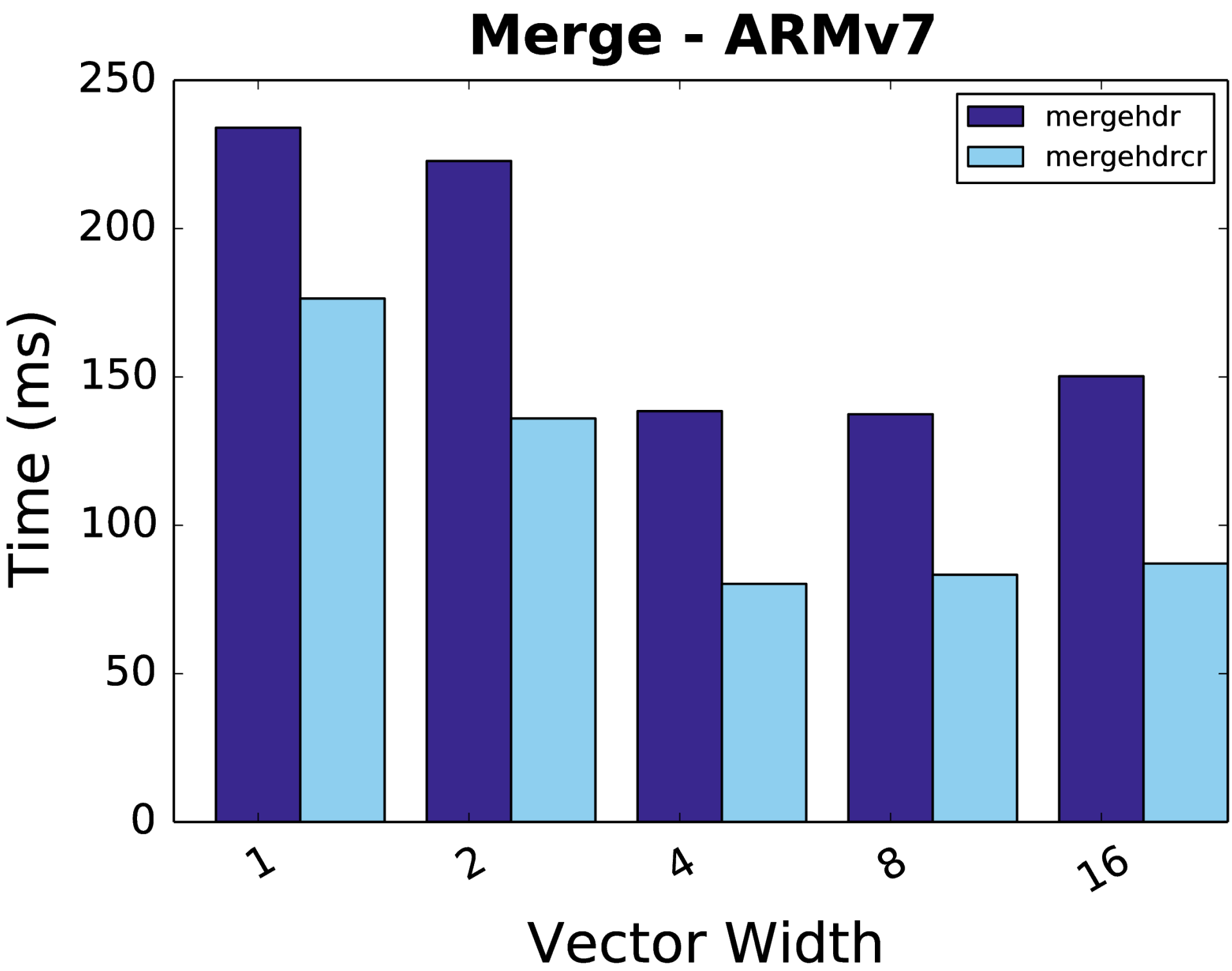

Align - AVX2
Time (ms)
0
3
6
9
12
15
1
2
4
8
16
Vector Width
f
sum
f$1
sum$1
f0

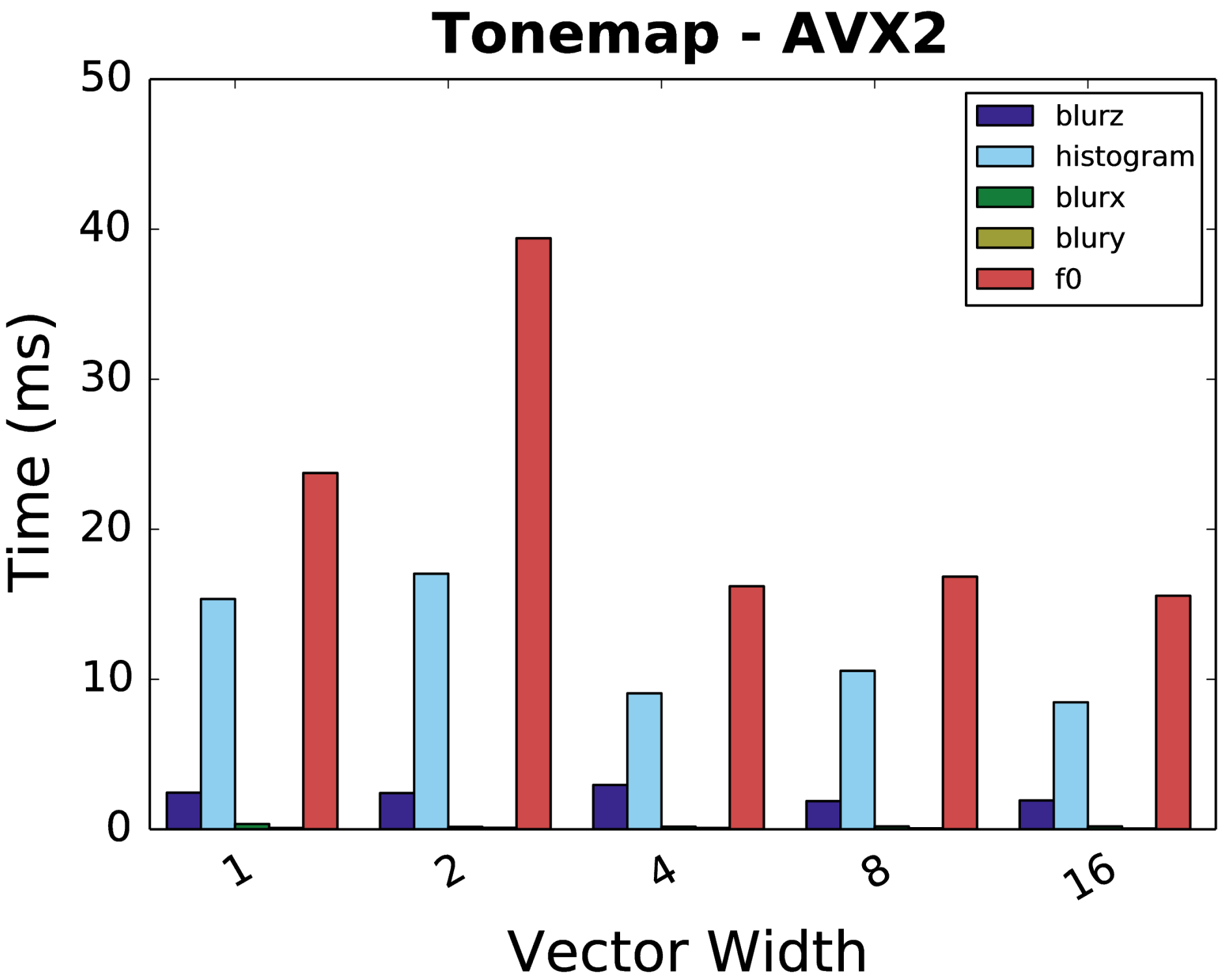

Align - ARMv8
Time (ms)
70
60
50
40
30
20
10
0
1
2
4
8
16
Vector Width
f
sum
f$1
sum$1
f0

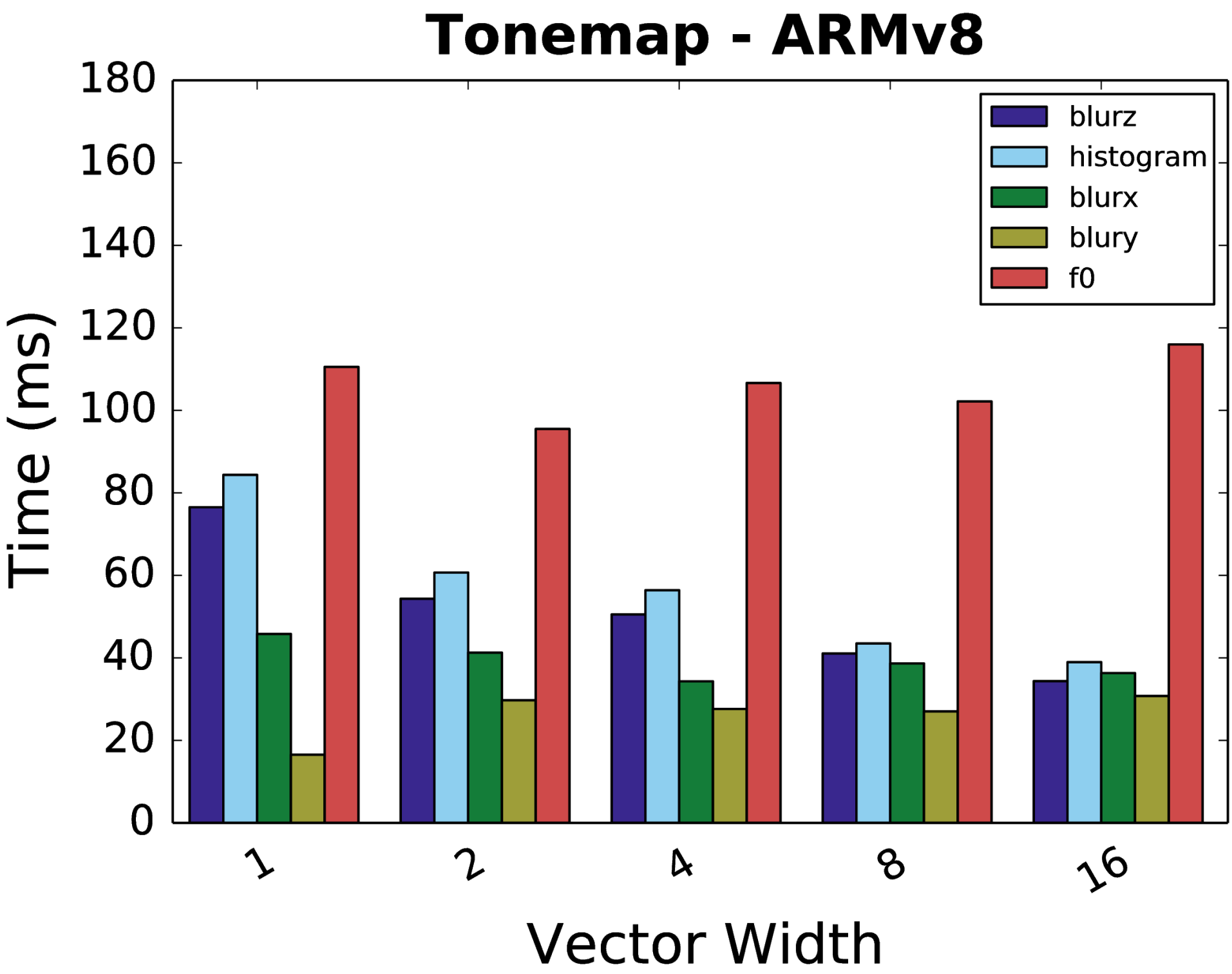

Align - ARMv7
Time (ms)
100
90
80
70
60
50
40
30
20
10
0
1
2
4
8
16
Vector Width
f
sum
f$1
sum$1
f0

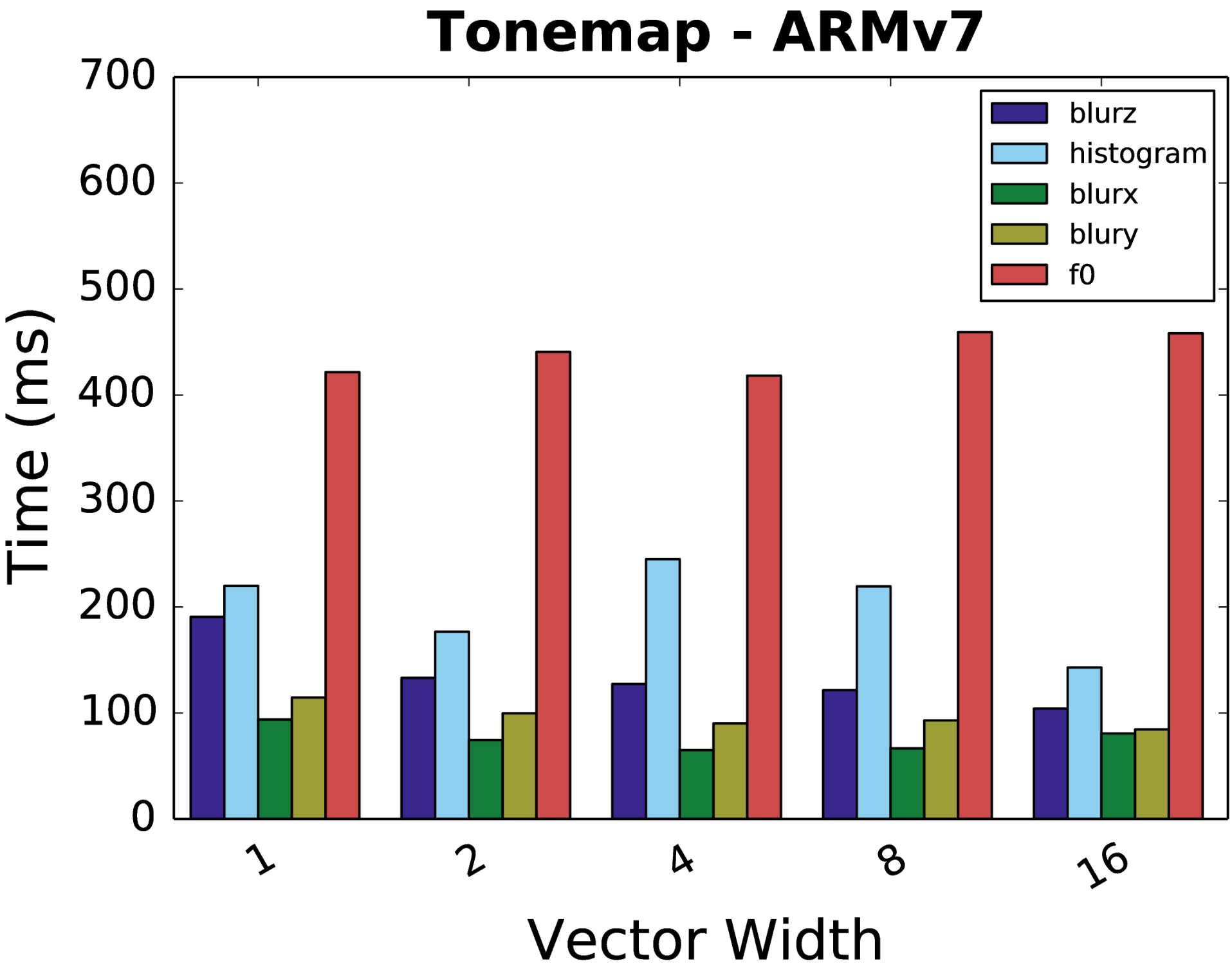

Merge - AVX2
mergehdr
mergehdrcr
Time (ms)
0
10
20
30
40
50
60
1
2
4
8
16
Vector Width


Merge - ARMv8
Time (ms)
280
240
200
160
120
80
40
0
mergehdr
mergehdrcr
1
2
4
8
16
Vector Width


Merge - ARMv7
250
200
150
100
50
0
Time (ms)
mergehdr
mergehdrcr
1
2
4
8
16
Vector Width


Tonemap - AVX2
Time (ms)
50
40
30
20
10
0
1
2
4
8
16
Vector Width
blurz
histogram
blurx
blury
f0


Tonemap - ARMv8
Time (ms)
180
160
140
120
100
80
60
40
20
0
1
2
4
8
16
Vector Width
blurz
histogram
blurx
blury
f0


Tonemap - ARMv7
Time (ms)
700
600
500
400
300
200
100
0
blurz
histogram
blurx
blury
f0
1
2
4
8
16
Vector Width

HDR Example

The complete Halide HDR pipeline
3 frames
12MP
800ms processing time


Image 1
Correctly Exposed


Image 2
Underexposed


Image 3
Overexposed


Output

Conclusions

Performance is generally proportional to number of hardware vector ALUs.
Works best if vector size is > 2.
No penalty for using a larger vector size than the natural vector size.
But:
Steep learning curve
Schedule optimization takes time
Not applicable to all domains

Fin