STORAGE DEVELOPER CONFERENCE

SDC 22

Fremont, CA
September 12-15, 2022

BY Developers FOR Developers

A SNIA Event

HDD Computational Storage Benchmarking

A Journey into Instruction per Cycle

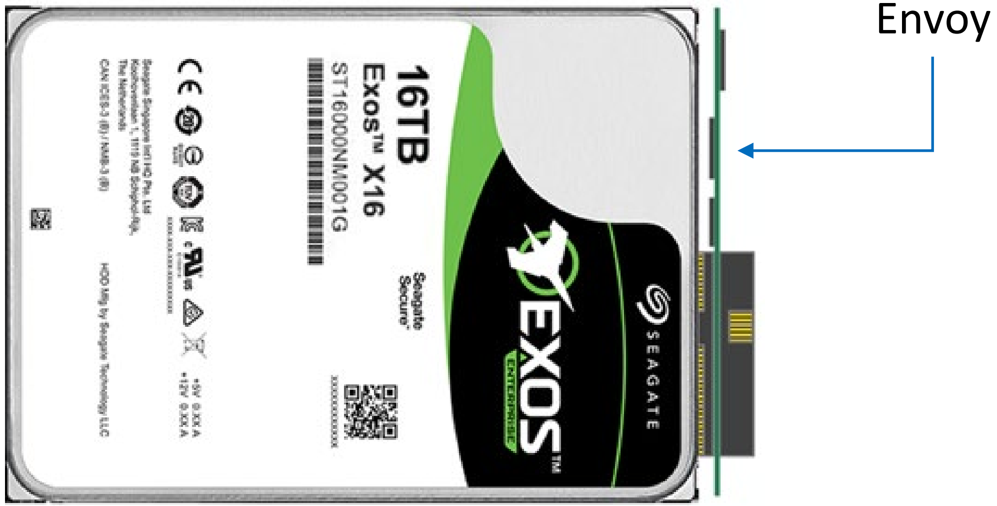
Presented by Philip Kufeldt, Seagate Technologies

Who Am I

- Philip Kufeldt, Technologist @ Seagate Technology
  - Focused on HDD Computational Storage research
  - Direct external research projects for Seagate
    - Human Cell Atlas (HCA) with UCSC
    - Campaign Storage 2.0 with LANL
  - I am NOT a CPU architecture expert
    - I did re-read my Hennessy and Patterson though (circa 1990)
  - Although Seagate is an ARM licensee, I have not had access to ARM licensee information
    - Everything in this presentation comes from public sources or experiments

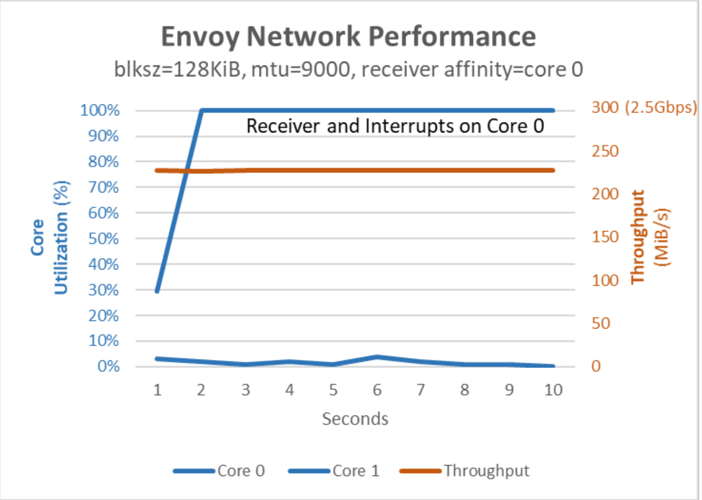
Envoy Interposer

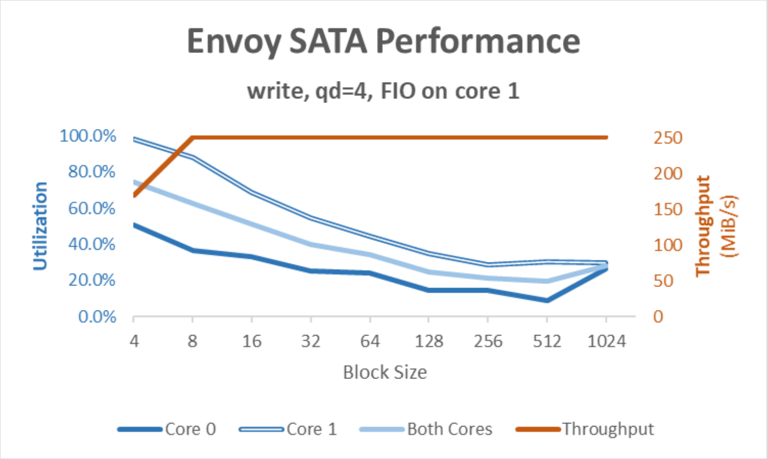
- Interposer card transforms a SATA HDD into a network attached computational storage device (CSD)
- Hardware
  - 1GHz Cortex A53 ArmV8 (Marvell Armada 88F3720)
    - 2 cores
    - 32KiB L1 I-cache and D-cache
    - 256KiB L2 Cache
    - Supports SIMD/Neon extensions
  - 1GiB DDR4 800Mhz 16-bit
  - Dual 2.5 Gbps Ethernet interfaces
  - 128 MiB NOR Flash
- Envoy runs Ubuntu 20.04LTS
  - a 5.16 Linux kernel
- Being deployed in HCA and CS2.0 research projects
  - computational storage functions will leverage the FPU

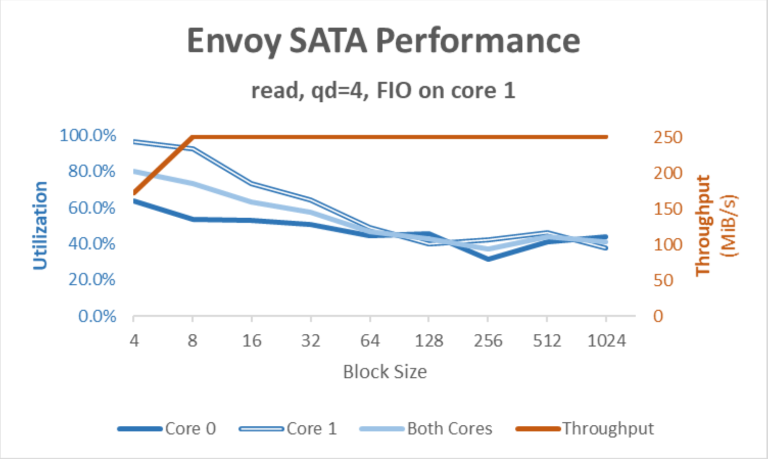


Standard Benchmarking

- Traditional storage and network benchmarks
  - Using fio and iperf3
  - No big issues







- However, application performance showed a problem

HCA T-Statistic

- HCA uses a T-Statistic to compare new sample data to existing archived data
  - Double precision data is in a matrix and the T-Statistic is calculated across columns
  - Computational storage functions performing this on the drive showed significant performance issues
- T-Statistic
  - Primary calculations are a running mean, means squares, sample variance and stddev.
  - Means and means squares are calculated incrementally across the data
    - variance and stddev calculated as a final step outside of main loop
  - Algorithm used is Welford’s as presented in Knuth’s Art of Programming, one loop is:

$$nmean = mean + \frac{(D_n \ - mean)}{n}$$

$$sqs = sqs + \left((D_n \ - mean) \times (D_n \ - nmean)\right)$$

$$mean = nmean$$

Where D_n is column data (e.g. D_{11}) and a separate mean and means squares is calculated per row.

$$\begin{bmatrix} D_{11} & \cdots & D_{n1} \\ \vdots & & \vdots \\ D_{1m} & \cdots & D_{nm} \end{bmatrix} \rightarrow \begin{bmatrix} means_1 \\ \vdots \\ means_m \end{bmatrix} \text{and} \begin{bmatrix} sqs_1 \\ \vdots \\ sqs_m \end{bmatrix}$$

Recreation

- A sample program was created to isolate the T-Statistic calculation and provide a test vehicle
  - This program
    - allocates a matrix and fills it with in with either zero’s or random data.
    - can set matrix sizes at runtime and perform a configurable number of calculation runs on the same data
    - can set data size
    - can set core affinity
  - The primary calculation function
    - allocates means and squares vectors
    - loops over the matrix performing the Welford calculations
    - it is single threaded
  - This program was constructed such that the compiler’s optimizer could leverage SIMD FPU units
    - AVX/SSE2 on Intel and NEON on ARM.
    - Compiled with -O3
    - Lessons learned about data layout to leverage gcc’s optimizer
  - Resulting data was validated against Excel model

Usage:

```
Usage: ./accumulate [OPTIONS]
Timed column means and means squares calculations
for a matrix, sized r x c.
Where, OPTIONS are [default]:
        -r rows          Number of Rows [3]
        -c cols          Number of Columns [3]
        -R runs          Number of Runs [3]
        -A core #        Set the affinity to core [no affinity]
        -e               Do not fill the matrix with data
        -t {i|f|d|s|a}   Set datatype to int, float, double,
                         double SIMD, or double assembler
        -v               Be verbose
        -?               Help
```

Recreation

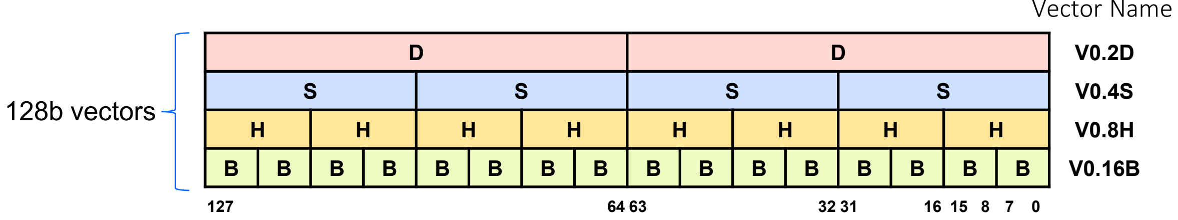
- A sample program was created to isolate the T-Statistic calculation and provide a test vehicle
  - This program
    - allocates a matrix and fills it with in with either zero's or random data.
    - can set matrix sizes at runtime and perform a configurable number of calculation runs on the same data
    - can set data size
    - can set core affinity
  - The primary calculation function
    - allocates means and squares vectors
    - loops over the matrix performing the Welford calculations
    - it is single threaded
  - This program was constructed such that the compiler's optimizer could leverage SIMD FPU units
    - AVX/SSE2 on Intel and NEON on ARM.
    - Compiled with -O3
    - Lessons learned about data layout to leverage gcc's optimizer
  - Resulting data was validated against Excel model

Pseudo Code:

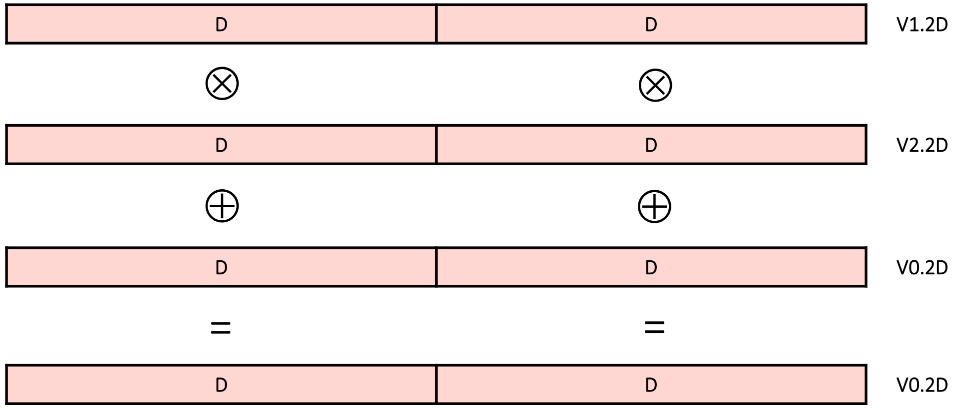
```
matrix = create_matrix(rows, cols);
means = create_vector(cols);
sqs = create_vector(cols);

// Handle initial column, col0
for (r=0; r<rows, r++) {
      means[r] = matrix[0, r];
      sqs = 0.0;
}

// Now the rest of the columns
for (c=1; c<cols; c++) {
      nc = c + 1;       // nc is col # starting from 1
      for (r=0; r<rows; r++) {
           dmean = (table[c][r] - means[r]);
           nmeans = means[r] + (dmean/nc);
           sqs[r] = sqs[r] + ((dmean) * (table[c][r] - nmeans));
           means[r] = nmeans;
      }
}
```

SIMD

ARM Neon 128bit FPU registers



Single Instruction Multiply and Accumulate MAC



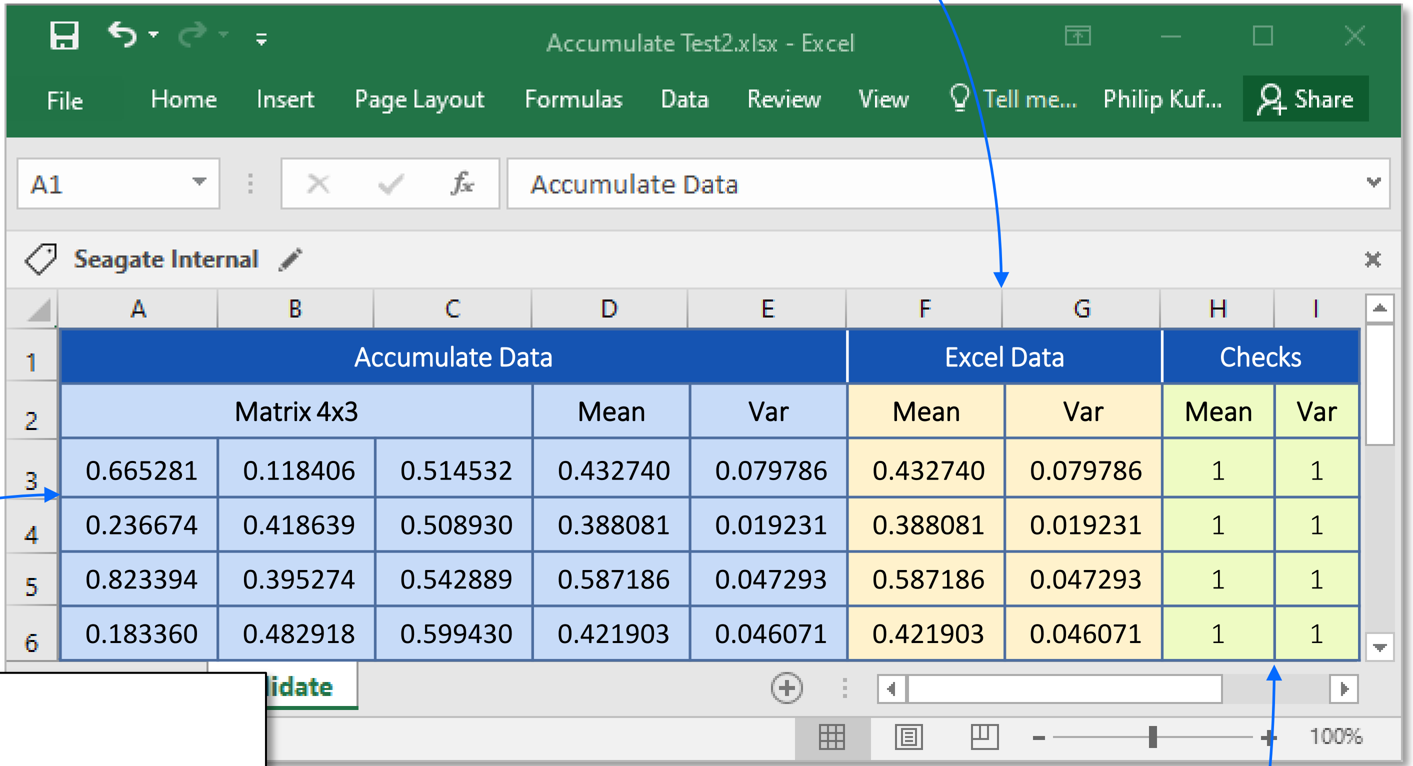
```
fmla v0.2d, v1.2d, v2.2d
```

C Code that Optimizes to SIMD code:

```
dmean = (double *)malloc(rows*sizeof(double));
nmeans = (double *)malloc(rows*sizeof(double));
means = (double *)malloc(rows*sizeof(double));
vars  = (double *)malloc(rows*sizeof(double));

if (!vars || !means || !nmeans || !dmean ) {
       printf("Alloc failed\n"); return(-1);
}

/* Initialize 1st column */
c=0;
for (r=0; r<rows; r++) {
       means[r] = table[c][r];
       vars[r]  = 0.0;
}
nc = 1;

/* Calulate the running means and variances */
for (c = 1; c<cols; c++) {
       nc++;

       for (r=0; r<rows; r++) {
              dmean[r]  = (table[c][r] - means[r]);
              nmeans[r] = means[r] + (dmean[r]/nc);

              vars[r] = vars[r] + ((dmean[r])*(table[c][r] - nmeans[r]));
              means[r]  = nmeans[r];
       }
}
```

Excel Model

Excel Calculated Values



Accumulate Data

	A	B	C	D	E	F	G	H	I
1	Accumulate Data					Excel Data		Checks	
2	Matrix 4x3			Mean	Var	Mean	Var	Mean	Var
3	0.665281	0.118406	0.514532	0.432740	0.079786	0.432740	0.079786	1	1
4	0.236674	0.418639	0.508930	0.388081	0.019231	0.388081	0.019231	1	1
5	0.823394	0.395274	0.542889	0.587186	0.047293	0.587186	0.047293	1	1
6	0.183360	0.482918	0.599430	0.421903	0.046071	0.421903	0.046071	1	1

Cut n Paste

```
$ ./accumulate -c 3 -r 4 -t d -R 1 -v
...
0.665281, 0.118406, 0.514532, 0.432740, 0.079786
0.236674, 0.418639, 0.508930, 0.388081, 0.019231
0.823394, 0.395274, 0.542889, 0.587186, 0.047293
0.183360, 0.482918, 0.599430, 0.421903, 0.046071
...
```

Quick Checks
(4 digit checks)

Analysis

- Recreation program used internal Linux clocks to determine timings for various actions, such as:
  - Matrix creation and fill time
  - Main means, squares calculations
- Linux perf was used to look at:
  - L1 Cache misses - big candidate for perf issues
  - Branch misses
  - Instructions per cycle
- Runs were done on Xeon-based VMs and Envoy
- Geometry chosen to mimic HCA data sizes ~1MiB

Observations

- $TT_{Xeon} \approx TT_{Envoy} \times \left(\frac{Cycles_{Envoy}}{Cycles_{Xeon}}\right) \times \left(\frac{Insn/cycle_{Envoy}}{Insn/cycle_{Xeon}}\right)$
  - Rough approximation
- L1 d-cache and branch misses uninteresting
- Linux perf and internal timings matched

Matrix: 360x360; Runs: 1000; Dataset size: ~1MiB

perf stat	Xeon	Envoy	delta
Total Time mS	**86**	**2,126**	**2,472%**
Cycles GHz	4.1	0.98	23.9%
Instructions/cycle	**3.57**	**0.44**	**12.3%**
Branch misses	0.53%	0.66%	124%
L1 d-cache misses	8.13%	0.10%	1.2%

Recreation extended

- These significant differences drove a deeper dive into Envoy
  - New calculation routines were created:
    - Arm Neon SIMD version
      - Explicitly call Neon SIMD primitives
    - Arm Assembler version
      - mix of C and assembler
    - These somewhat excluded the compilers optimizer from main loop
  - Neon version yielded similar results to standard optimized C
    - Generally can’t out optimize the optimizer
  - Assembler version was slower
    - Used primarily to look at per instruction slowdowns
    - Utilize GCC Extended ASM

Arm64 Neon SIMD Code

```
float64x2_t  tvec, mvec, dvec, nmvec, sqvec, nsqvec, ncvec;

/* Initialize 1st column */
c=0;
for (r=0; r<rows; r++) {
      means[r] = table[c][r];
      vars[r]  = 0.0;
}
nc = 1.0;

/* Calulate the running means and variances */
for (c = 1; c<cols; c++) {
      nc++;
      ncvec = vdupq_n_f64(nc);
      for (r=0; r<rows; r+=2) {
            // Load base vectors
            tvec  = vld1q_f64(&table[c][r]);
            mvec  = vld1q_f64(&means[r]);
            sqvec = vld1q_f64(&vars[r]);

            // Calculate dmean and nmean
            dmvec = vsubq_f64(tvec, mvec);            // dmv=tv-v
            nmvec = vdivq_f64(dmvec, ncvec);          // nmv=dmv/ncv
            nmvec = vaddq_f64(mvec, nmvec);           // nmv=mv+nmv

            // Calculate nsquares
            nsqvec = vsubq_f64(tvec, nmvec);          // dvv=tv-nmv
            nsqvec = vmlaq_f64(sqvec, nsqvec, dmvec); // nsqv=sqvec+(dvv*dmv)

            // Store results
            vst1q_f64(&means[r], nmvec);
            vst1q_f64(&vars[r], nsqvec);
      }
}
```

GCC Extended ASM

- What is this?
  - Support for inline assembly that utilizes C variables and labels
  - Allows the C code generator and optimizer to work with inline assembly
- How?

```
__asm__ asm-qualifiers ( AssemblerTemplate
                          : OutputOperands
                        [ : InputOperands
                        [ : Clobbers
                        [ : GotoLabels ] ] ] )

Where,
     asm-qualifiers = inline | volatile | goto
     AssemblerTemplate = assembly instructions
     In/Out Operands = [ [asm-name] ] constraint (c-expr)
     Clobbers = registers | memory
     GotoLabels = c-labels
```

- Used for assembly version of my T-Statistic code
  - Not going to explore all of the power of extended ASM
  - Find out more here:

Recreation extended

Assembler Code

```
for (c = 1; c<cols; c++) {
      /* optimization: Used in the row loop below */
      cspace = c * rows;

      /* optimization: setup a dup'ed vect of the num of cols */
      nc++; ncvec = vdupq_n_f64(nc);

      for (r=0; r<rows; r+=2) {
      __asm__ volatile (
            /* Register Usage:
             *    x0-x2 &table[c][r], &means[r], &sqs[r]
             *    x5-x7 table idx(ti), means idx(mi), sqs idx (vi)
             *    v10 table[c][r] vector (tv)
             *    v11 means[r] vector (mv)
             *    v12 sqs[r] vector (vv)
             *    v13 delta mean vector (dmv)
             *    v14 new mean vector (nmv)
             *    v15 delta var vector (dvv)
             *    v16 temp var vector (tv)
             *
             * The formula for the table index is:
             *    ((c * rows) + r) OR (cspace + r)
             */
/* 1 */     "add  x5, %[cspace], %[r]\n\t“      // ti = cspace + r
/* 1 */     "lsl  x5, x5,  #3\n\t"              // ti *= sizeof(double)
/* 1 */     "lsl  x6, %[r],  #3\n\t"            // mi = r*sizeof(double)
/* 1 */     "lsl  x7, %[r],  #3\n\t"            // vi = r*sizeof(double)
```

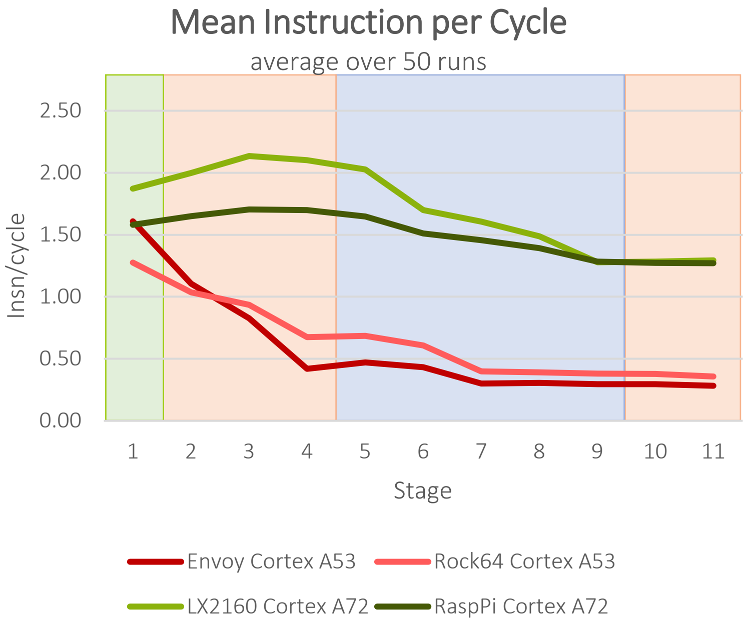
```
/* 1 */    "add  x0, x5, %[table]\n\t"   // table+mi
/* 1 */    "add  x1, x6, %[means]\n\t"   // means+mi
/* 1 */    "add  x2, x7, %[sqs]\n\t"     // sqs+vi

/* 2 */    "ldr  q10, [x0]\n\t"          // tv = table[c][r,r+1]
/* 3 */    "ldr  q11, [x1]\n\t"          // mv = means[r,r+1]
/* 4 */    "ldr  q12, [x2]\n\t"          // vv = sqs[r,r+1]

/* 5 */    "fsub v13.2d, v10.2d, v11.2d\n\t"         // dmv=tv-mv
/* 6 */    "fdiv v14.2d, v13.2d, %[ncv].2d\n\t"      // nmv=dmv/ncv
/* 7 */    "fadd v14.2d, v11.2d, v14.2d\n\t"         // nmv=mv+nmv
/* 8 */    "fsub v15.2d, v10.2d, v14.2d\n\t"         // dvv=tv-nmv
/* 9 */    "fmla v12.2d, v15.2d, v13.2d\n\t"         // vv+=dvv*dmv
/* 10 */   "str  q14, [x1]\n\t"          // means[r,r+1] = nmv
/* 11 */   "str  q12, [x2]\n\t"          // sqs[r,r+1] = vv
           "\n\t"
           : /* No Output Operands */
           : [table]  "r" (base_table),
             [means]  "r" (means),
             [sqs]    "r" (sqs),
             [r]      "r" (r),
             [cspace] "r" (cspace),
             [ncv]    "w" (ncvec)
           : "x0", "x1", "x2", "x5", "x6", "x7",
             "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16"
           );
      }
}
```

Recreation extended - Assembler Loop

- Permitted a vehicle to isolate pipeline stalls
  - By staging in instructions I could see their effect on the instruction per cycle metric.
    - Use ifdefs to add stages
  - 11 stages
    - Stage 1 are the first instructions that calculate memory indices, all scalar ALU ops.
    - Stages 2-4 are memory access instructions, loads.
    - Stages 5-9 are SIMD Vector FPU ops
    - Stages 10-11 are memory access instructions, stores.
  - After running it on the Envoy, I also had 3 other systems with ARMv8s to compare against
    - 4 Core Cortex A53 - Pine Rockpro64 - RK3399*
    - 16 Core CortexA72 - Solid Run HoneyComb - LX 2160A
    - 4 Core Cortex A72 - Raspberry Pi - BCM2711

Instruction per Cycle Comparison

		Instruction/cycle			
		CortexA53		CortexA72	
		Armada	RK3399	LX2160A	BCM2711
Stage	Description	Mean	Mean	Mean	Mean
1	Just matrix, means and vars address calculations: ALU Ops	1.61	1.28	1.87	1.58
2	Load matrix 128-bit value into q10: ldr	1.10	1.04	2.00	1.65
3	Load means 128-bit values into q11: ldr	0.83	0.93	2.13	1.70
4	Load vars 128-bit values into q12: ldr	0.42	0.68	2.10	1.70
5	Delta means = table value - means value: fsub	0.47	0.68	2.03	1.65
6	New Means = delta /num cols: fdiv	0.43	0.61	1.70	1.51
7	New Means = new means + means value: fadd	0.30	0.40	1.61	1.46
8	delta vars = table value - new means: fsub	0.30	0.39	1.49	1.39
9	vars = vars + (delta vars * delta means): fmla	0.30	0.38	1.28	1.28
10	Store means results: str	0.30	0.38	1.28	1.27
11	Store vars results: str	0.28	0.36	1.29	1.27

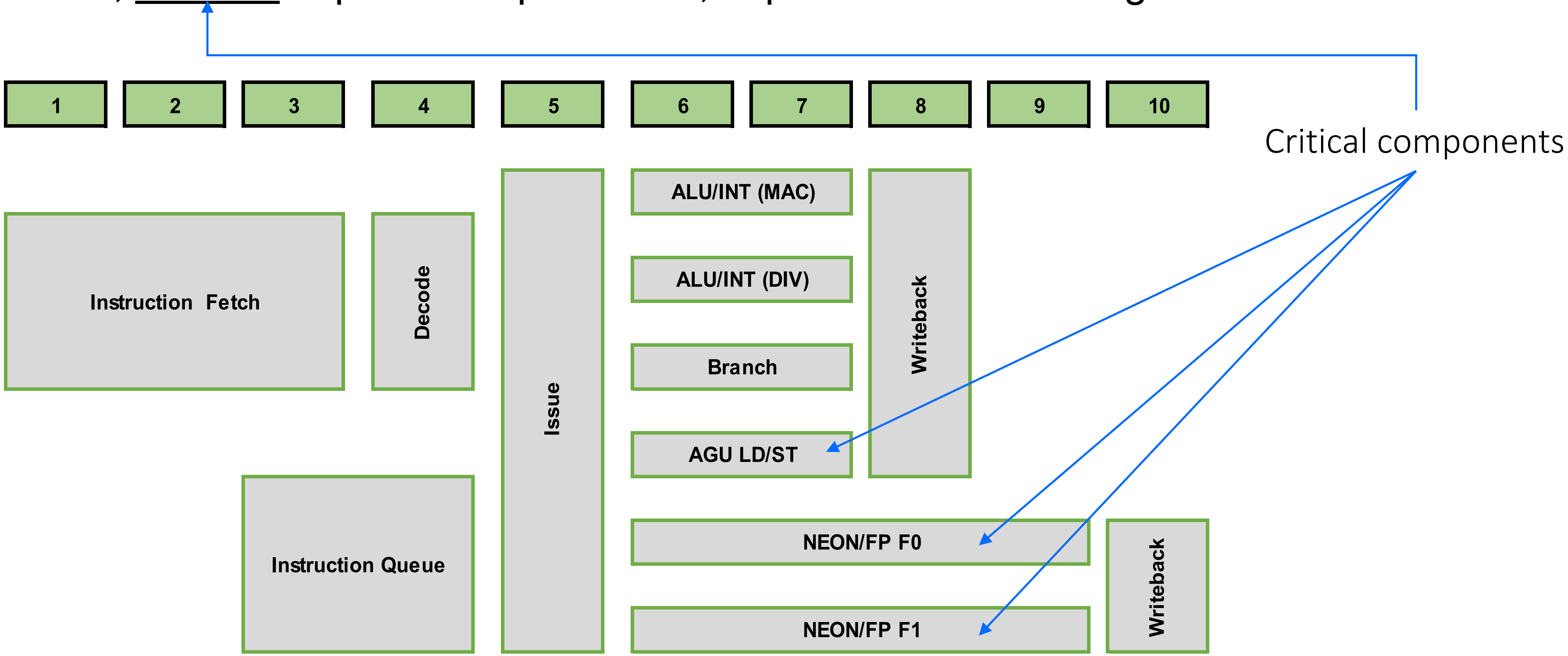


Observations

- Clear division between A53 and A72
- Memory accesses create the biggest drop for A53
  - A72's actually rise
  - Envoy's 16b DDR bus not the big culprit
- Still a significant drop in the vector ops
  - Possibly due to register contention

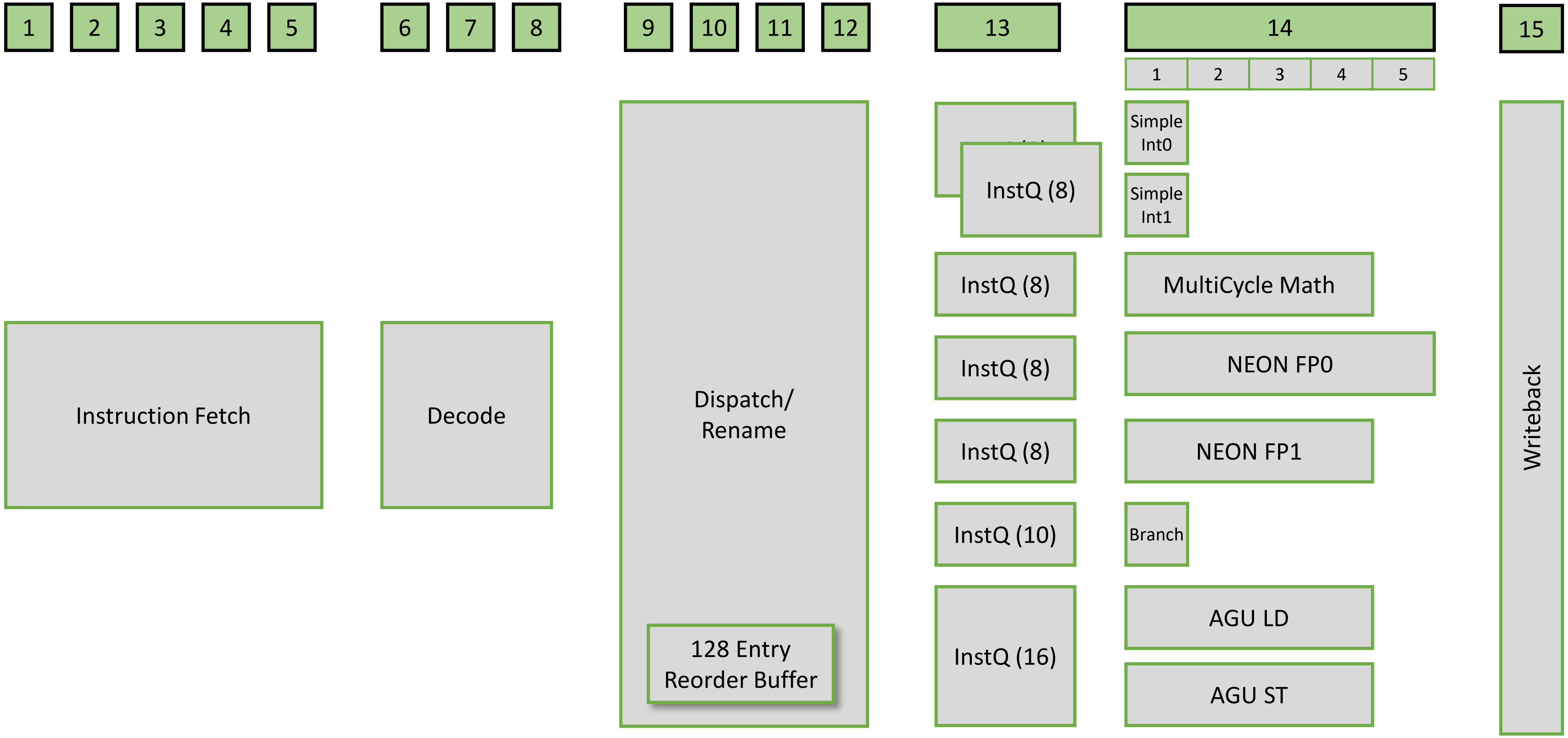
Cortex-A53 Pipeline

- 2-wide decode, in-order superscalar processor, capable of dual-issuing some instructions



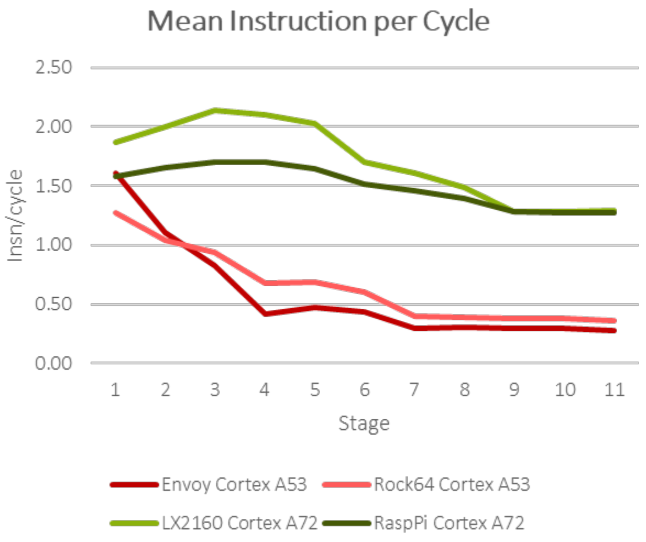
Cortex-A72 Pipeline

- 3-way decode, out-of-order superscalar pipeline

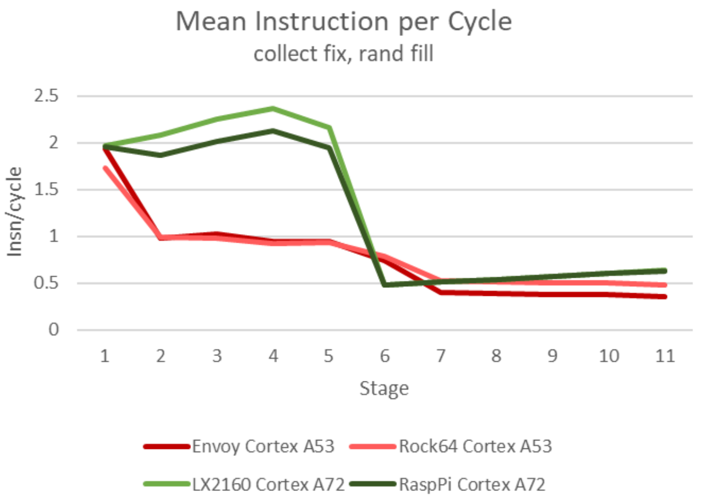


Post Script

- Graph was flawed in 2 ways
  1. Error in calculating insn/cycle means, not a significant error, but corrected
     - Incorrectly init-ed mean to 0 instead of first data point
  2. The matrices in the graph were zero filled, effect was significant



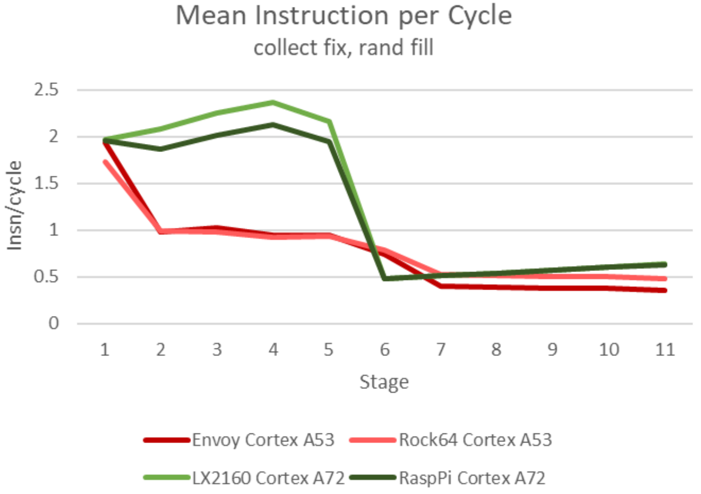
versus

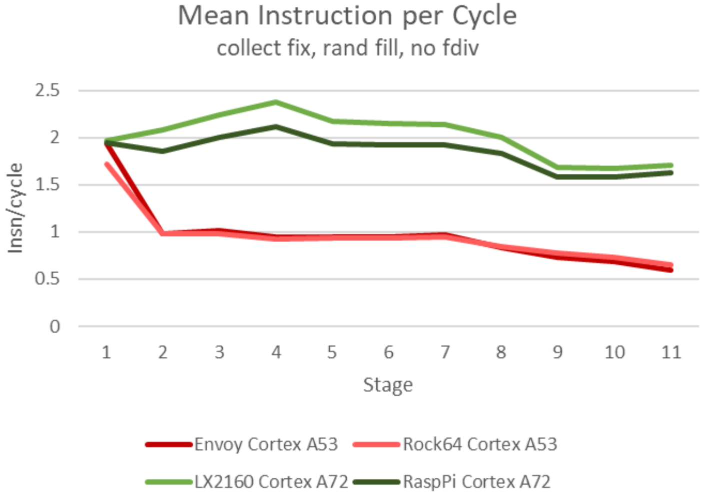


- What’s going on here?

Post Script

- Stage 6 is an fdiv
  - this is weighting-the-means calculation for the current # of columns
- Looks like FPU has an optimized path for a zero numerator
- Divides are not viewed as important for performance
  - FPU divides are always high latency
  - Multiply and accumulate (MAC) is the most important
  - There may be only one FP unit capable
    - Area efficiency issues
  - Divides can block/stall the pipeline until complete
  - SIMD divides may actually be done serially.
- C optimizers remove divides where possible
- I converted the code to multiply the inverse

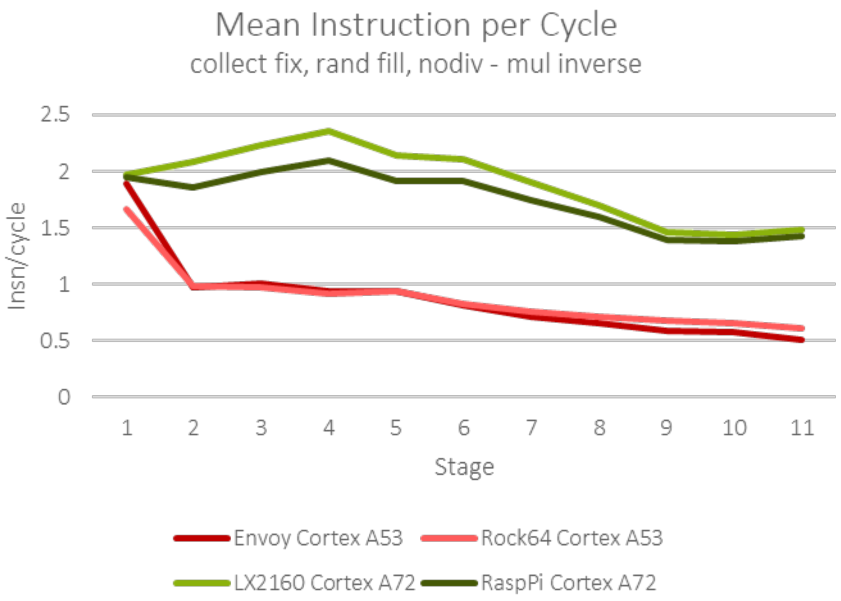

Mean Instruction per Cycle
collect fix, rand fill


Mean Instruction per Cycle
collect fix, rand fill, no fdiv

Post Script

- Valid data chart
  - Divide replaced with a multiply of inverse
  - Done once per column

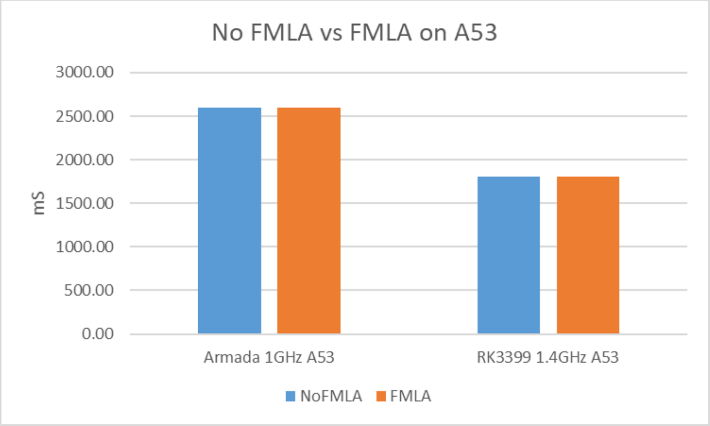
```
ncvec = vdupq_n_f64(nc);

    changed to

ncvec = vdupq_n_f64(1.0/nc);
```

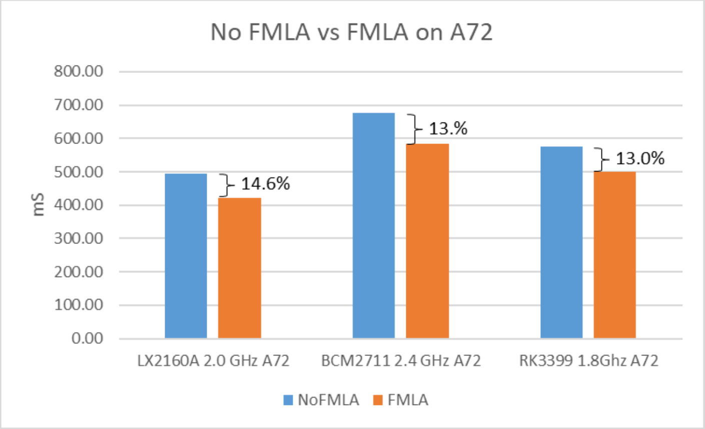


- Further refinement to merge stage 6 and stage 7
  - One MAC instruction
  - Stage 6 a noop

Post Post Script

- Conversion to from fdiv to fmul allowed another fmla (MAC)
- Welford has two opportunities for fmla
  - Decided to quantify the benefit of one fmla vs fmul+fadd across both A53 and A72
  - Create binaries with both approaches





  - A53 there is no benefit
    - indicates fmla implementation is simply fmul+fadd
  - A72 gets a 13.0-14.6% bump

Conclusions

- Pipelines are important
    - Looking at A53 vs A72
        - 25x to 6x
- FPU implementations are important
    - just because you code a MAC doesn't mean you get a MAC
- Would like to see slow down limited to <5x
    - to make $$ reduction a viable argument
- Need to look at a new Envoy
    - A72 or A55
    - DDR Bus may not be a critical requirement

Matrix: 360x360; Runs: 1000; Dataset size: ~1MiB

perf stat	Xeon	Envoy A53	LX2160 A72	RaspPi A72	RK3399* A53/72
Total Time mS	86	2,126	456	524	661
Cycles GHz	4.1	0.98	1.97	1.48	1.39
Instructions/cycle	3.57	0.44	1.03	1.20	0.34
Branch misses	0.53%	0.66%	NA	NA	16.01%
L1 dcache misses	8.13%	0.10%	0.14%	1.10%	3.19%

*Don't fully trust RK3399 results

Please take a moment to rate this session.

Your feedback is important to us.

If you have any questions: philip.kufeldt@seagate.com