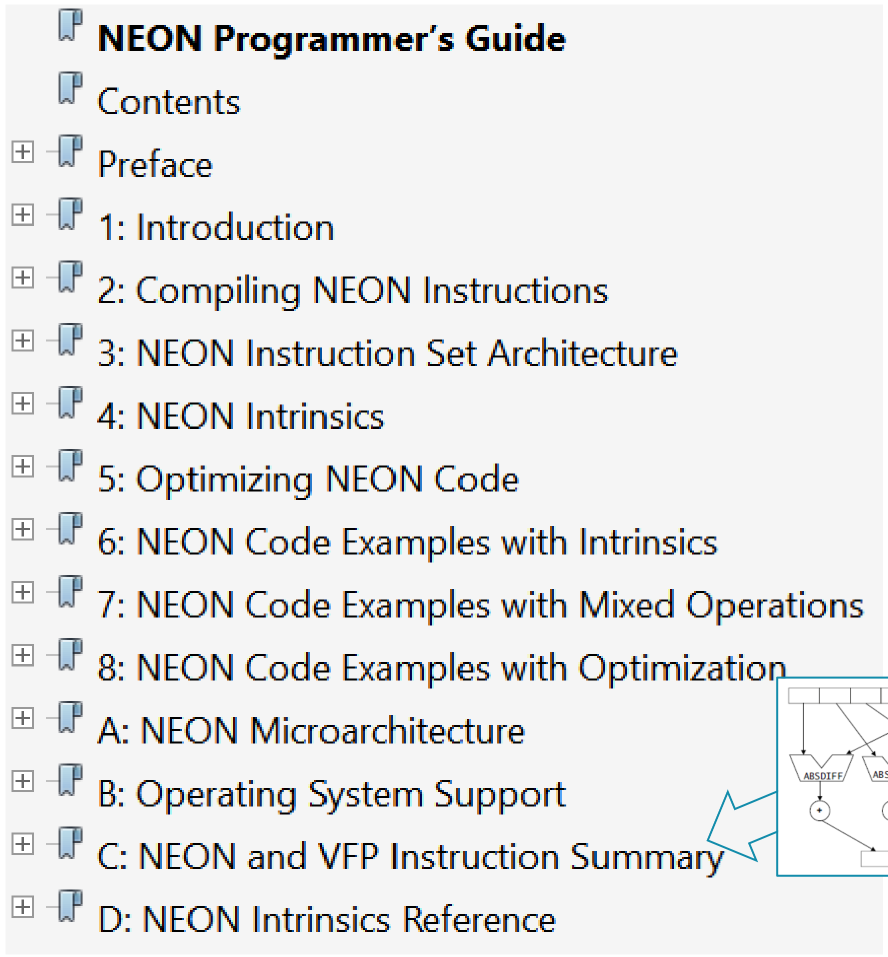
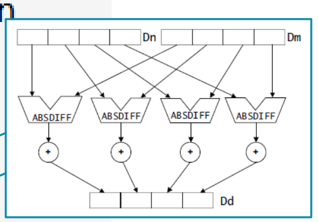
Using NEON Advanced SIMD Processing

References

- NEON Programmer's Guide DEN0018 (NPG) – read this first!





- Instr. Functionality: ARM Arch. Ref. Manual
  - Load/Store: 4.11
  - Register Transfer: 4.12
  - Data Processing: 4.13, 4.14
- ARM C Language Extensions IHI0053 (ACLE)
- ARM NEON Intrinsics Reference IHI0073 (NIR)
- Performance: Cortex-A72 Software Optimization Guide UAN0016

NC STATE UNIVERSITY

USING THE ASIMD INSTRUCTIONS

Again, Do Instruction Set Architectures Matter?

- Online discussion by Jack Ganssle, Bill Gatliff, Niall Murphy, and Jim Turley at Embedded.com

- No!
  - C compiler hides differences and emulates missing features with code
    - Native word size, floating point math, subroutine call penalty, conditional branch delay
  - Most compilers don't ***use*** those great fast instructions
    - Table lookup and interpolate, 3D matrix operations, etc
  - As long as the processor runs fast enough, costs dominate
- Yes!
  - Use "intrinsics" (inline assembly code) to use fast instructions
  - Code density depends on processor
  - It takes time and money to come up to speed on a new architecture, so go with what gets you a product sooner
  - More engineers available for hiring if you use a common architecture
  - If no (or just buggy) tools are available, it's not worth using

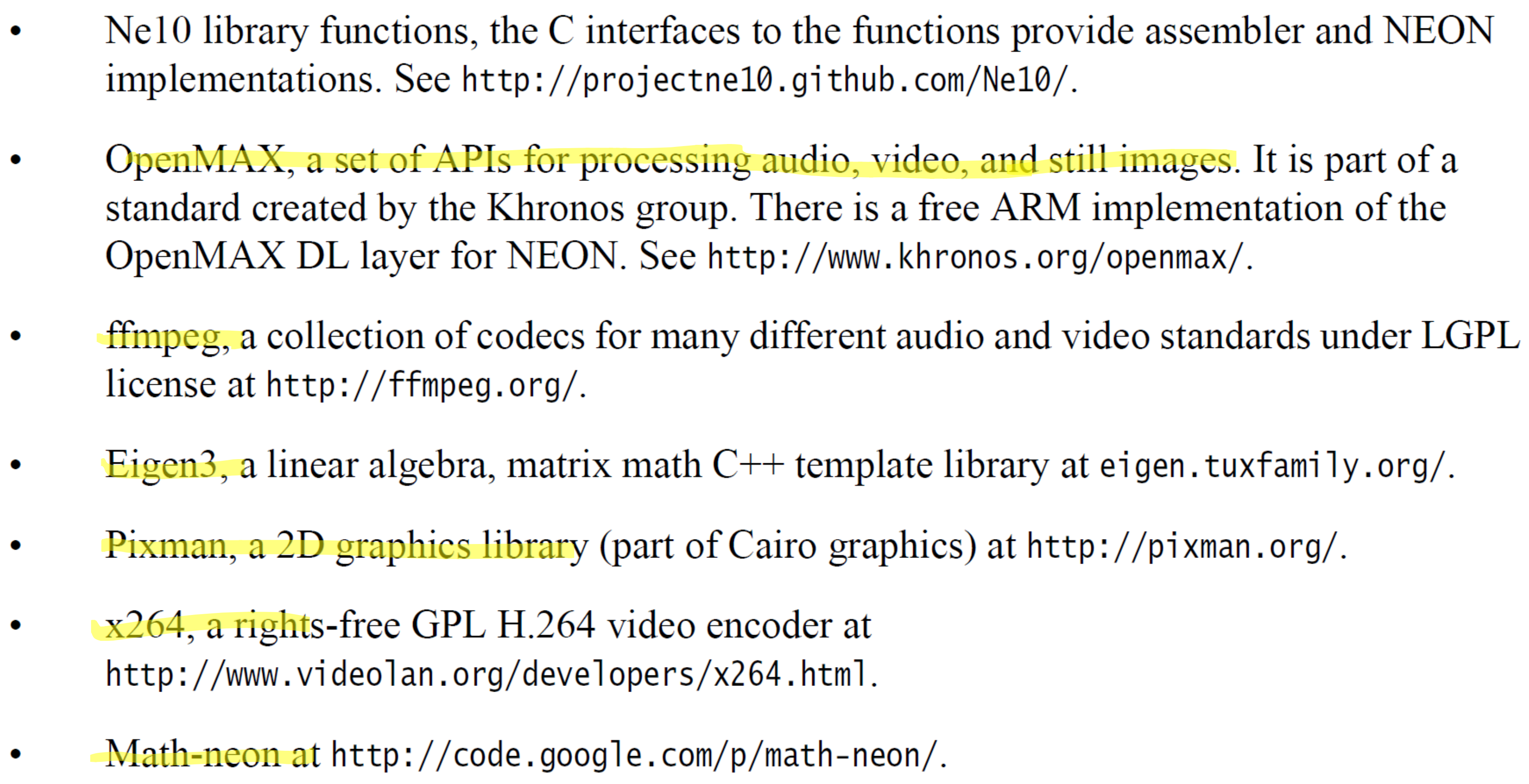
NC STATE UNIVERSITY

How Can We Use These SIMD Instructions?

- Write C code, call functions from **SIMD libraries**
  - Need NEON-optimized libraries for your application
- Write **C code, rely on the compiler** to generate SIMD instructions
  - Depends on compiler's ability to vectorize code
  - "How can I get the compiler to do what I want?"
- Write **C code with compiler intrinsics** to specify SIMD instructions
  - Provides more control and takes care of many details
  - Need clear understanding of data layout and processing flow
- Write a separate **SIMD assembly code** module, link it with our C code
  - Provides full control but you must manage all the details
  - Need clear understanding of data layout and processing flow
  - See "Getting Better Object Code"

USING NEON LIBRARIES

Many NEON Libraries Available



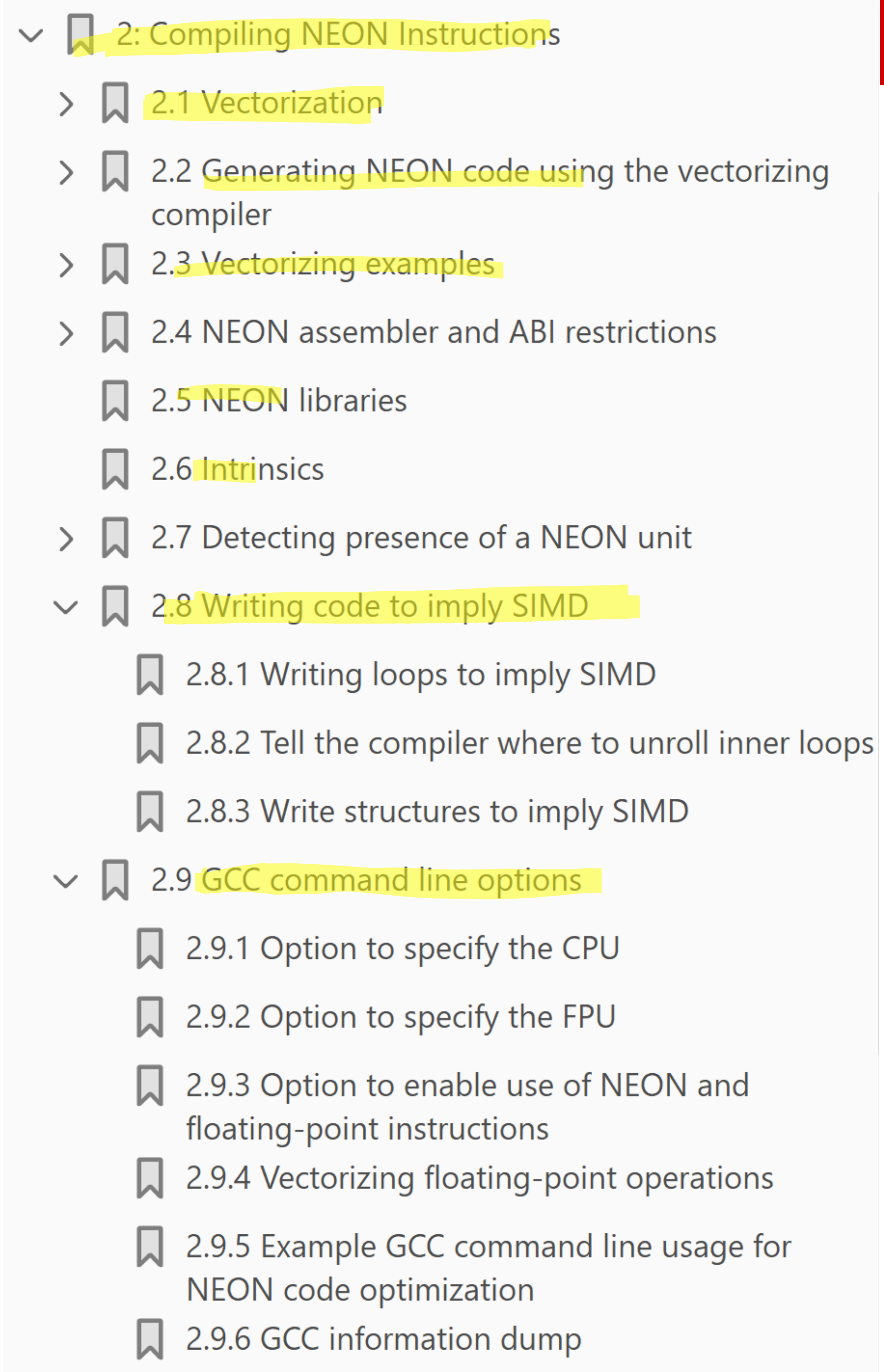
- Ne10 library functions, the C interfaces to the functions provide assembler and NEON implementations. See `http://projectne10.github.com/Ne10/`.
- OpenMAX, a set of APIs for processing audio, video, and still images. It is part of a standard created by the Khronos group. There is a free ARM implementation of the OpenMAX DL layer for NEON. See `http://www.khronos.org/openmax/`.
- ffmpeg, a collection of codecs for many different audio and video standards under LGPL license at `http://ffmpeg.org/`.
- Eigen3, a linear algebra, matrix math C++ template library at `eigen.tuxfamily.org/`.
- Pixman, a 2D graphics library (part of Cairo graphics) at `http://pixman.org/`.
- x264, a rights-free GPL H.264 video encoder at `http://www.videolan.org/developers/x264.html`.
- Math-neon at `http://code.google.com/p/math-neon/`.

- From NEON Programmer's Guide, DEN0018A
- And search for "neon-optimized libraries"

HELPING THE COMPILER

Documentation

- NPG Chapter 2:
- And…
  - NEON Support in Compilation Tools: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0004a/ch01s01s01.html
  - Introducing NEON: http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf



NC STATE UNIVERSITY

Helping GCC Make Fast Code

- CPU specification
  - **-mcpu=cortex-a72**
  - **-mfpu=crypto-neon-fpu-armv8**
- Vectorization – More details shortly
  - **-ftree-vectorize** (enabled with **–O3**)
    - Enables vectorization (both loop and basic-block)
    - Defaults to 64-bit NEON double registers (Dn)
  - **-mvectorize-with-neon-quad**
    - Targets 128-bit NEON quad registers (Qn)
  - **-funsafe-math-optimizations**
    - Treat all summation variables as reduction variables. More later…
- Other
  - **-fsingle-precision-constant**
    - Treat floating-point constants as single-precision, not double-precision
  - **-ffast-math**
    - NEON floating point math uses Flush-to-Zero mode, not compliant with IEEE-754
    - This flag tells compiler it doesn't need to generate IEEE-754-compliant code
  - **-Ofast**
    - Enables all –O3 optimizations and **-ffast-math**, **-fallow-store-data-races** and **fno-protect-parens**

Linkage Methods

- How are floating-point subroutine arguments/return values passed?
  - In ARM registers (r0-r3)? Software linkage
  - In FPU and NEON registers? Hardware linkage

- Command line options
  - soft: uses software linkage, and all floating-point operations are calls to library functions
  - softfp: uses software linkage, but allows compiler to generate hardware floating-point instructions
  - hard: uses hardware linkage and allows compiler to generate hardware floating-point instructions

- **-mfloat-abi=hard**

Runfast Mode

- Allows some VFP instructions to execute in NEON unit
  - FADDS, FSUBS, FABSS, FNEGS, FMULS, FNMULS, FMACS, FNMACS, FMSCS, FNMSCS, FCMPS, FCMPES, FCMPZS, FCMPEZS, FUITOS, FSITOS, FTOUIS, FTOSIS, FTOUIZS, FTOSIZS, FSHTOS, FSLTOS, FUHTOS, FULTOS, FTOSHS, FTOSLS, FTOUHS, FTOULS

```
void enable_runfast() {
        static const unsigned int x = 0x04086060;
        static const unsigned int y = 0x03000000;
        int r;
        asm volatile (
                "fmrx %0, fpscr \n\t" //r0 = FPSCR
                "and %0, %0, %1 \n\t" //r0 = r0 & 0x04086060
                "orr %0, %0, %2 \n\t" //r0 = r0 | 0x03000000
                "fmxr fpscr, %0 \n\t" //FPSCR = r0
                : "=r"(r)
                : "r"(x), "r"(y)
        );
}
```

- Applicable to Cortex-A8. Does it still apply for Cortex-A72?

NC STATE UNIVERSITY

More GCC Flags

- **-ffinite-math-only**
  - There will be no overflows or results that are equivalent to infinity in the code, enabling more optimizations.

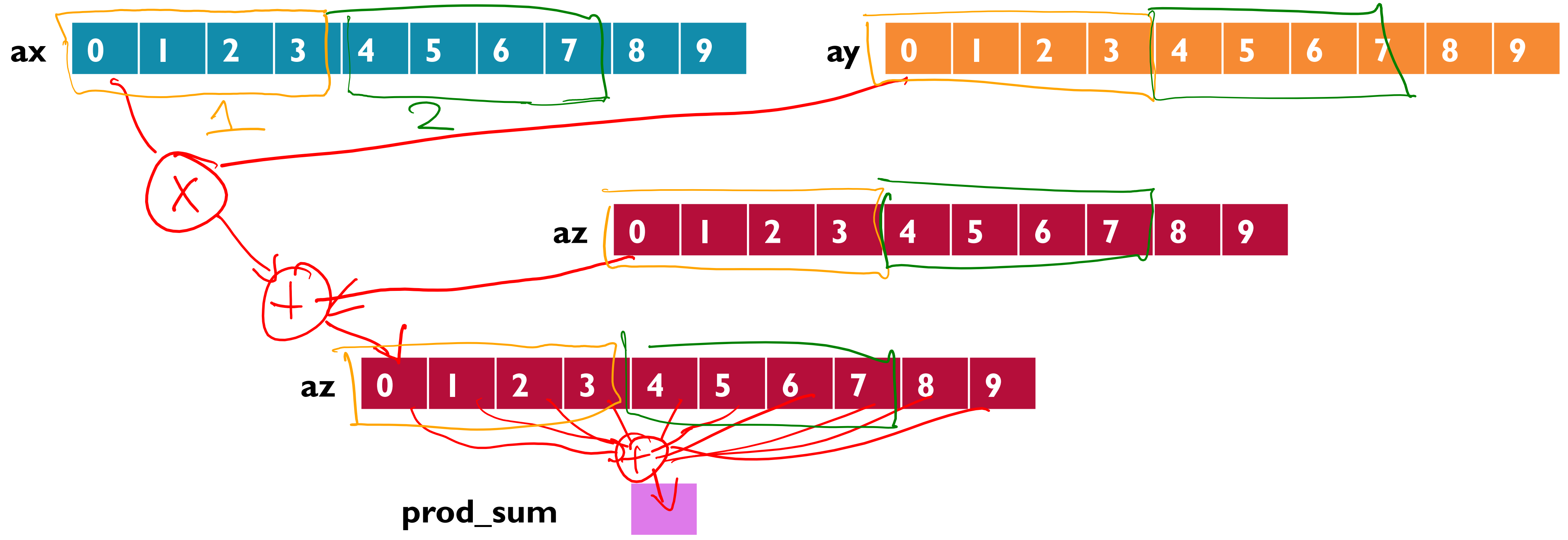
- **-fno-math-errno**
  - Eliminate all math error handling/generation code. Functions such as the **sqrt()** generate math errors when appropriate, and this can prevent inlining of such functions

BASICS OF VECTORIZATION

Want SIMD? Help Compiler Vectorize the Code

- Background
  - Scalar code: operates on one set of operands at a time
  - Vector code: operates on multiple sets of operands at a time
  - Vectorization: converting code from scalar to vector form
- Vectorization is **main** compiler optimization enabling use of SIMD instructions
  - Others possible, but don't work on as much code, harder to implement in compiler
- Best to try to vectorize loops first
  - Innermost loops often dominate execution time
  - Arrangement of instructions and data make vectorization easier (than the general case, e.g. straight-line code)
- Vectorization of loops is built on loop unrolling
- *Next:*
  - *Basic methods for loop unrolling*
  - *Command-line options for compiler*
  - *Coding practices*

Example Program: Neon0



- Per element: Multiply ax and bx, add product to az
- Sum all resulting az elements, return as prod_sum

```
int mult_ints(int * ax, int * ay, int * az, int n)
{
  int prod_sum = 0;
  for (int i=0; i < n; i++) {
    az[i] += ax[i]*ay[i];
    prod_sum += az[i];
  }
  return prod_sum;
}
```

What Does the Compiler Do With the Code?

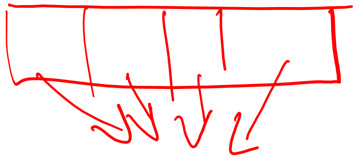
- Build with -O3 optimization

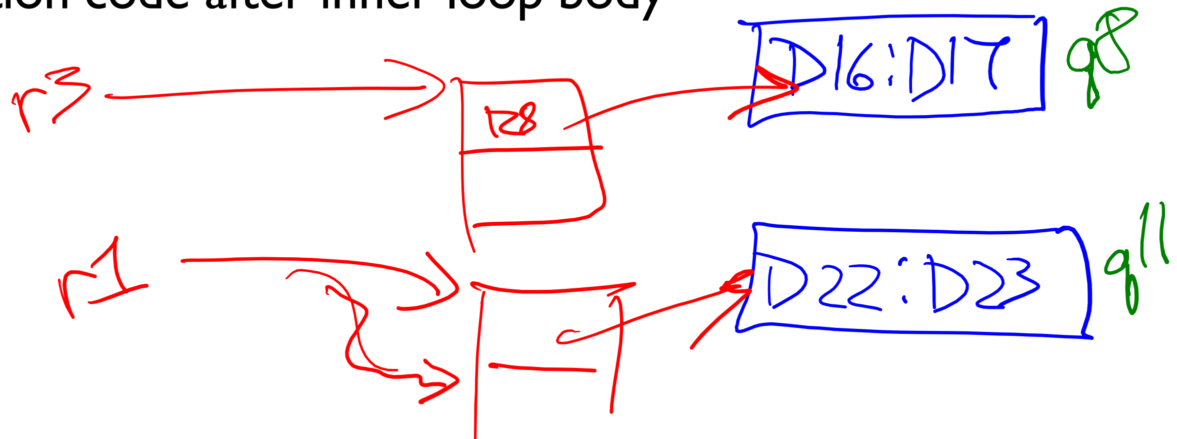
```
pi@raspberrypi:~/AES-2020/Speed/Vector/Neon0 $ sudo perf record ./neon0
Sum = 1829808256
Average 1.031 ns (1.547 cycles) per element (10000)
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.165 MB perf.data (4128 samples) ]
```

- Very fast! 1.547 cycles per element
- What's the compiler doing? Examine object code
  - Inner loop (c8): Vectorized loop body
  - Reduction code after inner loop body

```
int prod_sum = 0;
for (int i=0; i < n; i++) {
  az[i] += ax[i]*ay[i];
  prod_sum += az[i];
}
```

```
Samples: 4K of event 'cpu-clock', 0 Hz,
main  /home/pi/AES-2020/Speed/Vector/Ne
Percent          mov      r5, #0
  0.02   b4:     vmov.i32 q9, #0   ; 0x000
                 movw     r3, #704          ;
                 movt     r3, #9
                 mov      r1, r7
                 mov      r2, r6
 44.26   c8:     vld1.32 {d16-d17}, [r3]
 23.34           vld1.32 {d22-d23}, [r1]!
 27.56           vld1.32 {d20-d21}, [r2]!
                 vmla.i32 q8, q11, q10
  4.75           vst1.32 {d16-d17}, [r3]!
                 cmp      r0, r3
                 vadd.i32 q9, q9, q8
                 bne      c8
  0.07           vadd.i32 d18, d18, d19
                 subs     r4, r4, #1
                 vpadd.i32 d18, d18, d18
                 vmov.32 r3, d18[0]
```

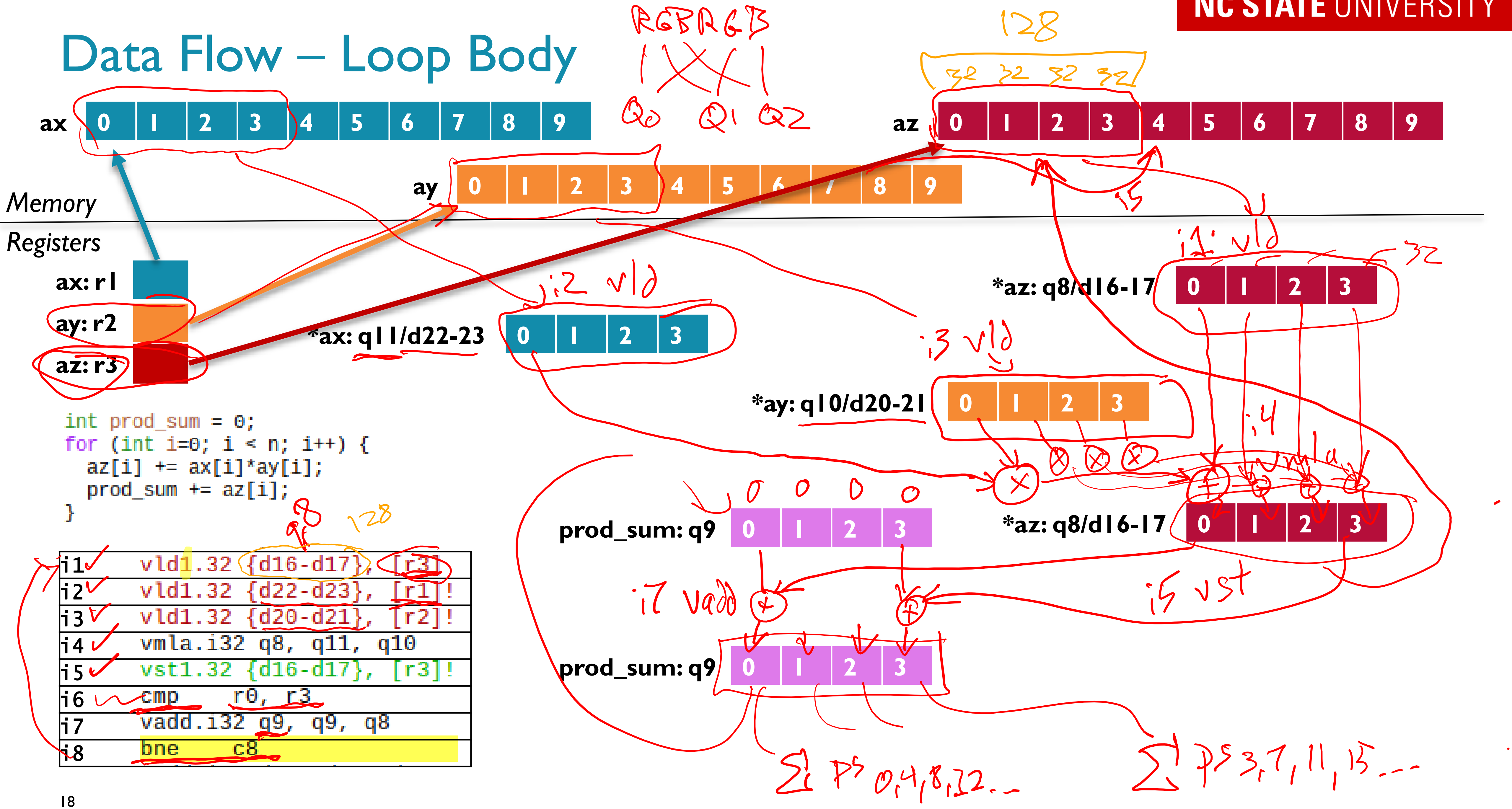




prod_sum

Red. code

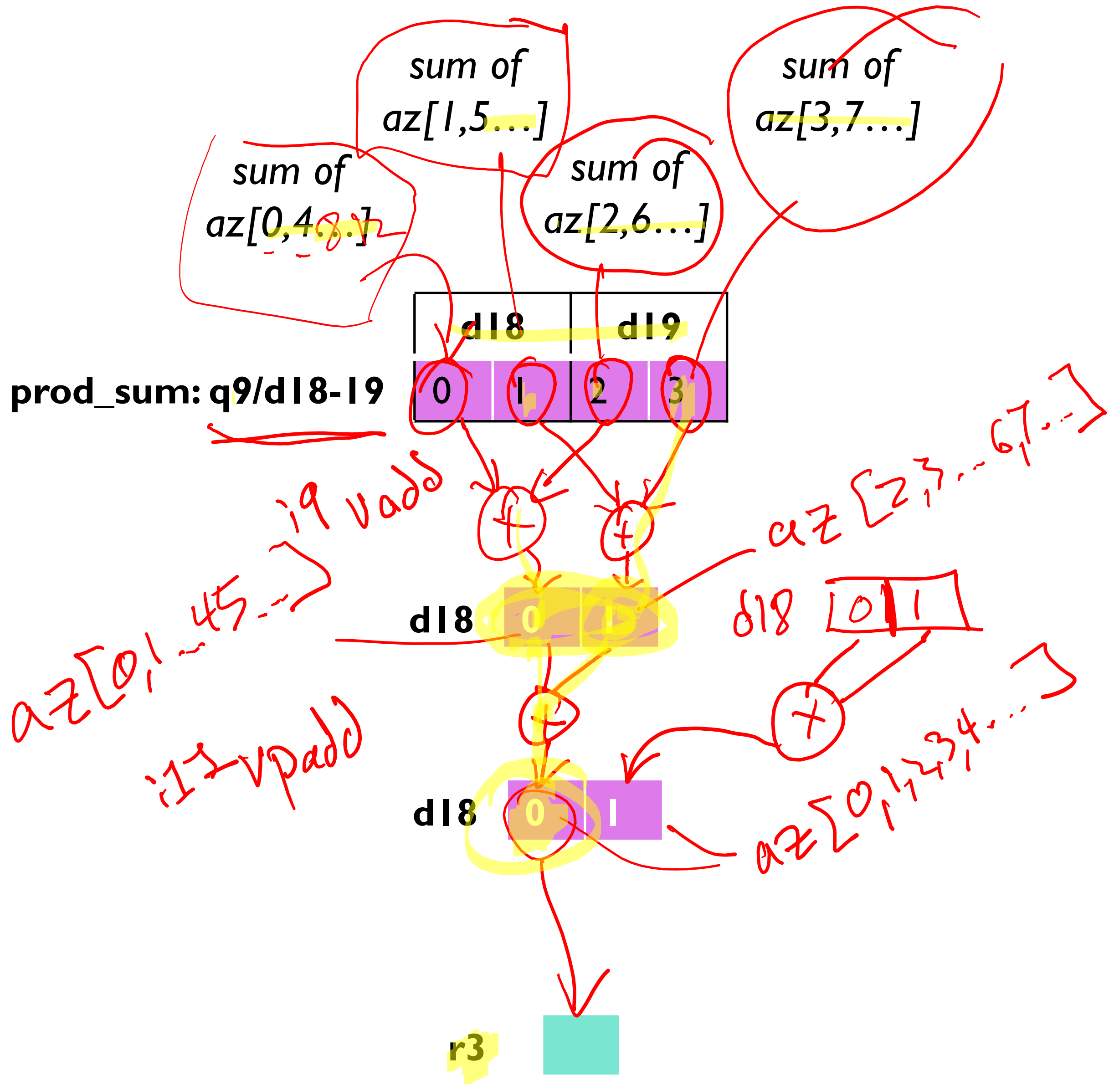
Data Flow – Loop Body



```
int prod_sum = 0;
for (int i=0; i < n; i++) {
  az[i] += ax[i]*ay[i];
  prod_sum += az[i];
}
```

i1✓	vld1.32 {d16-d17}, [r3]
i2✓	vld1.32 {d22-d23}, [r1]!
i3✓	vld1.32 {d20-d21}, [r2]!
i4✓	vmla.i32 q8, q11, q10
i5✓	vst1.32 {d16-d17}, [r3]!
i6	cmp r0, r3
i7	vadd.i32 q9, q9, q8
i8	bne c8

Data Flow – Reduction for prod_sum

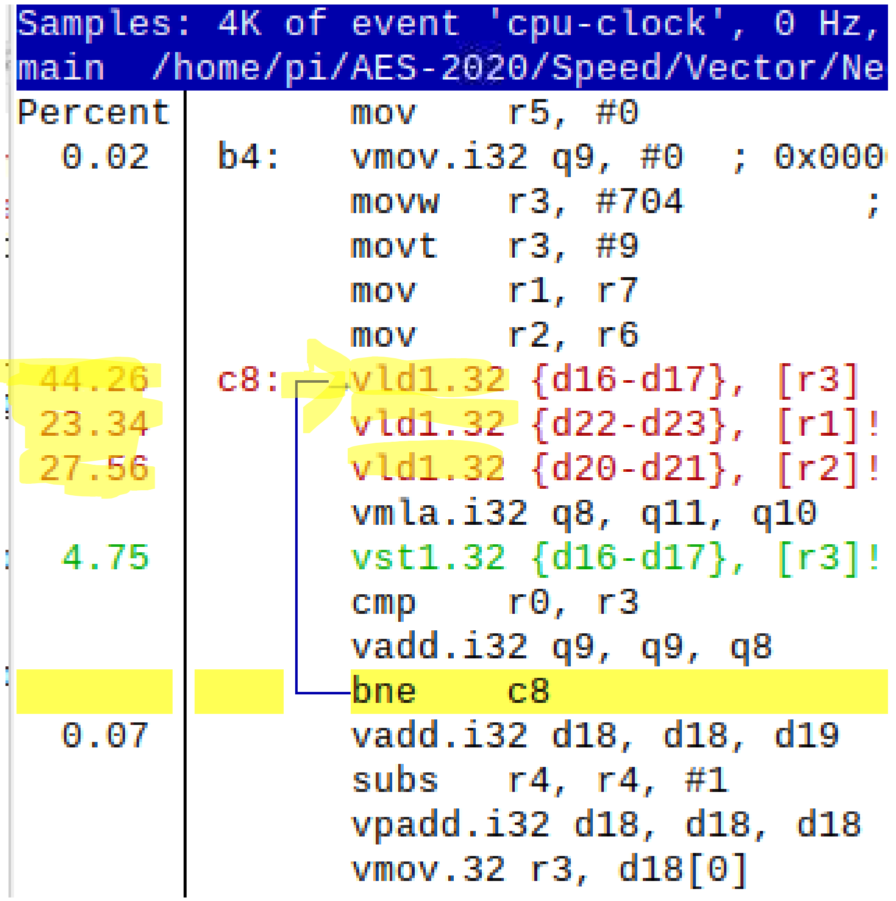


```
int mult_ints(int * ax, int * ay, int * az, int n)
{
  int prod_sum = 0;
  for (int i=0; i < n; i++) {
    az[i] += ax[i]*ay[i];
    prod_sum += az[i];
  }
  return prod_sum;
}
```

i9	vadd.i32 d18, d18, d19
i10	subs r4, r4, #1
i11	vpadd.i32 d18, d18, d18
i12	vmov.32 r3, d18[0]

Reflections

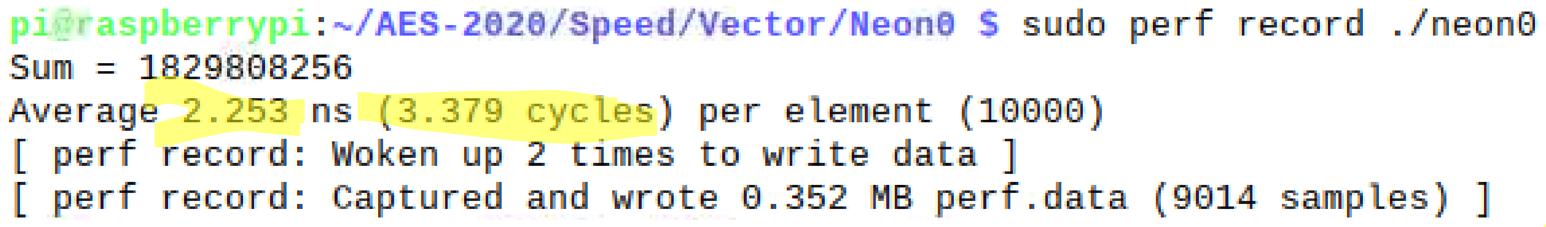
- Fast! Only 8 instructions execute to process 4 sets of array elements
- Performance
  - Most time (95.16%) is spent waiting for three load instructions
  - => Memory-bound program
- Best-case aspects of this example
  - Data is in separate arrays, easy to load into registers
  - Compiler can optimize and eliminate general-case code
    - Fixed iteration count
    - Vector size of 4 cleanly divides iteration count
  - No control flow in loop body
  - No data dependencies between loop iterations



```
Samples: 4K of event 'cpu-clock', 0 Hz,
main  /home/pi/AES-2020/Speed/Vector/Ne
Percent │       mov     r5, #0
   0.02 │ b4:   vmov.i32 q9, #0   ; 0x000
        │       movw    r3, #704           ;
        │       movt    r3, #9
        │       mov     r1, r7
        │       mov     r2, r6
  44.26 │ c8:┌─→vld1.32 {d16-d17}, [r3]
  23.34 │    │  vld1.32 {d22-d23}, [r1]!
  27.56 │    │  vld1.32 {d20-d21}, [r2]!
        │    │  vmla.i32 q8, q11, q10
   4.75 │    │  vst1.32 {d16-d17}, [r3]!
        │    │  cmp     r0, r3
        │    │  vadd.i32 q9, q9, q8
        │    └──bne     c8
   0.07 │       vadd.i32 d18, d18, d19
        │       subs    r4, r4, #1
        │       vpadd.i32 d18, d18, d18
        │       vmov.32 r3, d18[0]
```

What If? Disable Vectorization

- -fno-tree-vectorize

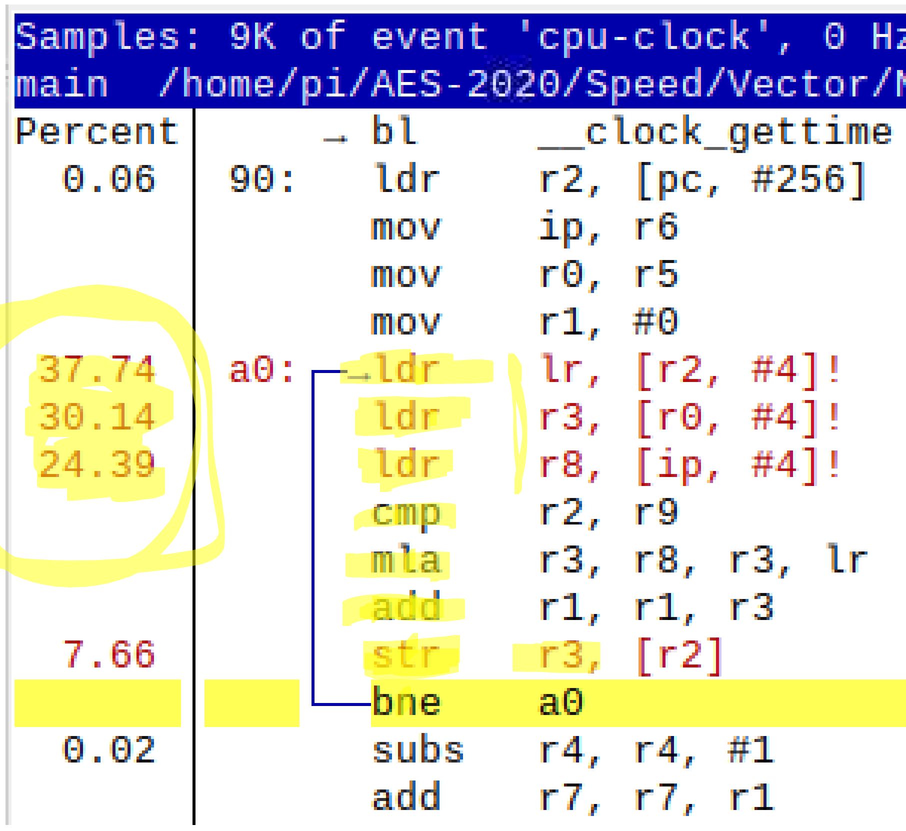


```
pi@raspberrypi:~/AES-2020/Speed/Vector/Neon0 $ sudo perf record ./neon0
Sum = 1829808256
Average 2.253 ns (3.379 cycles) per element (10000)
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.352 MB perf.data (9014 samples) ]
```

- Not as fast: 3.379 cycles

- Still have 8 instructions in inner loop, but they only process one set of array elements at a time

- Why is vectorized version only 3.379/1.547=2.18 times faster, despite processing 4x data per loop iteration?



```
Samples: 9K of event 'cpu-clock', 0 Hz
main  /home/pi/AES-2020/Speed/Vector/N
Percent        → bl      __clock_gettime
  0.06   90:   ldr     r2, [pc, #256]
               mov     ip, r6
               mov     r0, r5
               mov     r1, #0
 37.74   a0:┌→ldr     lr, [r2, #4]!
 30.14      │  ldr     r3, [r0, #4]!
 24.39      │  ldr     r8, [ip, #4]!
            │  cmp     r2, r9
            │  mla     r3, r8, r3, lr
            │  add     r1, r1, r3
  7.66      │  str     r3, [r2]
            └──bne     a0
  0.02         subs    r4, r4, #1
               add     r7, r7, r1
```

BASICS OF LOOP UNROLLING FOR VECTORIZATION

Why Understanding Loop Unrolling Matters

- Compiler tries to unroll loops when optimizing
  - Let's help the compiler by providing better code
  - We may need to tweak code to enable loop unrolling
- We may want to manually unroll loops
  - If compiler can't
  - If not using the compiler

Basic Loop Unrolling Process

```
// Original loop
for (i=0; i<n; i += 1) {
        sum_val += x[i];
}
```

```
// Unrolled loop
for (i=0; i < n-(F-1); i += F) {
        sum_val += x[i];
        sum_val += x[i+1];
        ...
        sum_val += x[i+(F-1)];
}

// Loop for remaining iterations
for (; i < n; i++) {
        sum_val += x[i];
}
```

- Unroll loop body
  - New loop body will perform F iterations of original loop body
  - Modify loop control code
    - Test: confirm at least F more iterations remain
    - Increment: Scale update by factor of F
  - Unroll loop by factor of vector size
    - Modify data processing instructions: Make F-1 copies of loop body instructions
    - Update references to data: Add 1 to F-1 to data value indices. May update pointers by factor of F.
- Create clean-up loop
  - Implement remaining iterations with non-unrolled code
  - No initialization of i

Loop Iteration Count Considerations

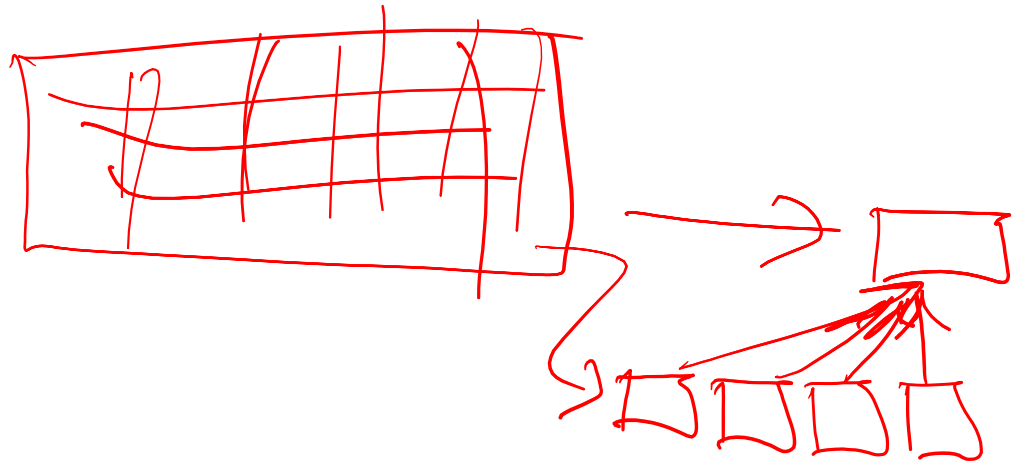
- Unrolling a loop with L iterations by a factor of F
  - **Unrolled loop** performs floor(L/F) iterations of the unrolled loop (performing F times as much work per iteration)
    - This unrolled loop will later be **vectorized**
  - **Clean-up loop** performs L modulo F remaining iterations of the original loop (performing 1x work per iteration)

- Compiler must generate code which operates correctly regardless of whether L is a multiple of F or not
  - Typically involves generating code to determine if there are at least F more iterations of work to perform
  - Can be simplified if compiler can determine if L is a multiple of F

NC STATE UNIVERSITY

Basic Loop Unrolling and Vectorization Process

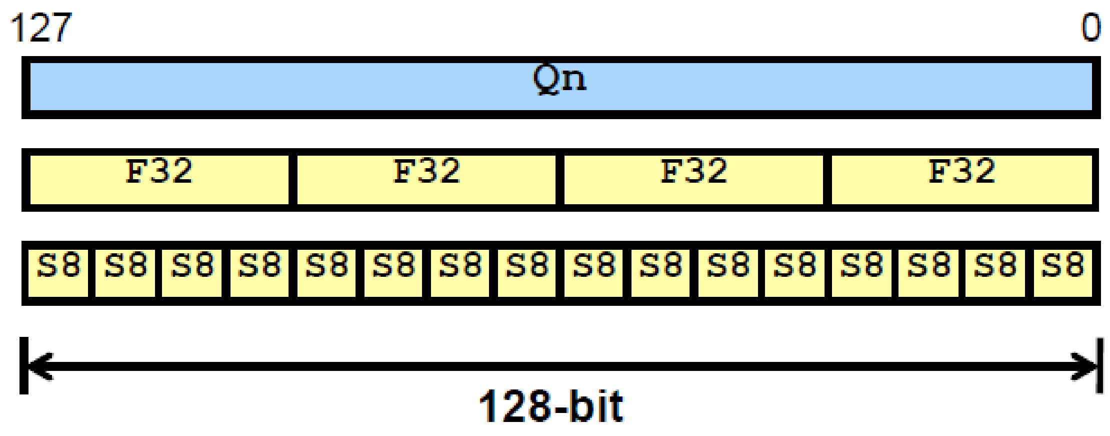
- Create prelude
  - Create vector values (and loop-independent variables) from scalars
- Unroll loop body
  - Modify loop control code
    - Test: confirm at least F more iterations remain
    - Increment: Scale update by factor of F
  - Unroll loop by factor of vector size
    - Modify loop body data processing instructions
      - If Unrolling: Make F-1 copies of instructions
      - If Vectorizing: replace each scalar instruction with a vector instruction
    - Update references to data: Add 1 to F-1 to data value indices. May update pointers by factor of F.
- Create postlude
  - Reduce (gather, condense, sum) data from vector to scalar form
- Clean-up
  - Implement remaining iterations with non-vectorized code



Selecting Good Loops

- Select an inner-most loop
  - With data in arrays
  - Without
    - Subroutine calls
    - Conditional control flow
    - Data dependencies within F successive loop iterations

- Determine loop unroll factor (= vector size) F
  - NEON registers 128 bits wide, options are:
    - 4 element vector of words
    - 8 element vector of half-words
    - 16 element vector of bytes



HELPING THE COMPILER WITH SIMD AND VECTORIZATION

NC STATE UNIVERSITY

Guidance for Making Code More Easily Vectorizable

- Refer to NPG 2.1.10
- Use short, simple loops
- Don't use `break` to exit loops
- Make loop iterations a power of two
- Let compiler know number of loop iterations
- Inline all functions called within the loop to vectorize
- Use arrays with indexing instead of pointers
- Don't use indirect addressing (multiple indexing or dereferencing)
- Use `restrict` to indicate that pointers don't reference overlapping areas of memory

Want SIMD? Write Code to Imply SIMD

- NPG, Section 2.8

- Write loops to imply SIMD
  - Use contents of structure in a single loop.
  - Improves cache performance.

```
for (...) { outbuffer[i].r = ....; }
for (...) { outbuffer[i].g = ....; }
for (...) { outbuffer[i].b = ....; }
```

```
for (...)
{
  outbuffer[i].r = ....;
  outbuffer[i].g = ....;
  outbuffer[i].b = ....;
}
```

- Tell compiler where to unroll inner loops
  - https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html
  - `#pragma GCC ivdep` There are no loop-carried dependencies preventing concurrent execution
  - `#pragma unroll` *n* Loop should be unrolled *n* times

Remember To Use The GCC Vectorization Flags

- **-ftree-vectorize**
  - Enables vectorization
    - **-ftree-loop-vectorize** loop vectorization
    - **-ftree-slp-vectorize** basic-block vectorization
  - Defaults to NEON double register (D), 64 bits long
  - –O3 implies –ftree-vectorize
- **-mvectorize-with-neon-quad**
  - Targets NEON quad register (Q), 128 bits long
- **-ftree-vectorizer-verbose**=<level>
  - <level> can range from 1 to 6
  - At 6, provides extensive (excessive?) information on vectorization attempts and barriers
- **-funsafe-math-optimizations**
  - Treat all summation variables as reduction variables. This assumption eliminates the inherent **loop-carried-dependencies** for such variables, thus allowing vectorization.

Constraining Loop Iteration Count

```
int accumulate(int * c, int len)
{
    int i, retval;

    for(i=0, retval = 0; i < (len & ~3) ; i++) {
        retval += c[i];
    }

    return retval;
}
```

Source: DHT 0004A, ARM Ltd.

- Example: What if len is always a multiple of four?
- Can tell compiler by masking off two LSBs of len in loop test
  - len & ~3 = len & 0x11111111...111100
- There are no remaining iterations to keep compiler from vectorizing the loop

NC STATE UNIVERSITY

Avoid Loop-Carried Dependencies

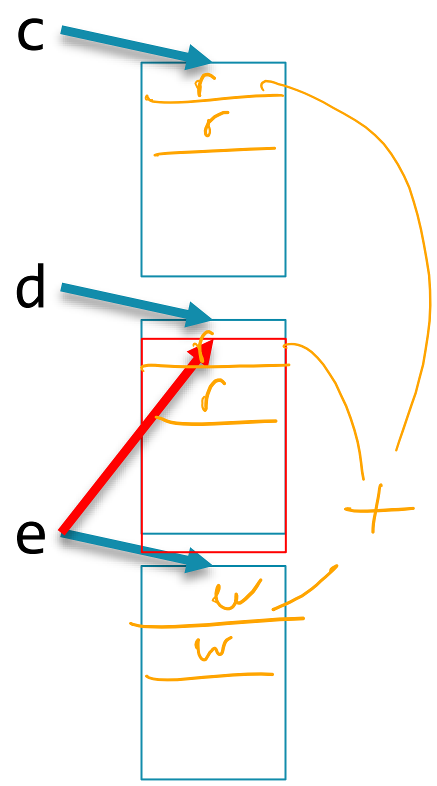
- Loop-carried dependency exists if a calculation in iteration *j* depends on the result of any previous iteration *i*, where *i*<*j*
- This dependency prevents vectorization
    - Can't do multiple iterations simultaneously
- Sometimes is possible to restructure code to remove it, but not always

```
float x[N], y[N];

for (n=1; n<N; n++) {
        x[n] = y[n] * x[n-1];
}


// Unrolling once leads to this
for (n=1; n<N; n+=2) {
        x[n] = y[n] * x[n-1];
        x[n+1] = y[n+1] * x[n];
}
```

Use restrict Keyword

```
int accumulate2(char * c, char * d, char * restrict e, int len)
{
    int i;

    for(i=0 ; i < (len & ~3) ; i++) {
        e[i] = d[i] + c[i];
    }

    return i;
}
```

Source: DHT 0004A, ARM Ltd.



- Read from c and d, write to e
  - e[i] depends on c[i] and d[i]
- What if e and d point to overlapping arrays?
  - e[i] might also be an element in d (e.g. d[i], d[i+1]…)
  - Order of operations may change with vectorization
  - Compiler can't vectorize safely, so it won't

- Tell compiler that the location accessed by *p* is not accessed by any other pointer within the current scope
  - Use *restrict* (C99 keyword) to describe a pointer *p*
  - GCC also supports __restrict and __restrict__

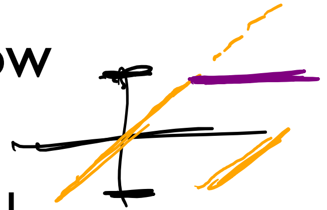
Use Appropriate Data Types

- In the ARM integer core, 8-bit operations are slower than 32-bit operations
  - Need code to extract byte from register before operation, extend it, and merge it back in after operation
- So, promote shorter data up to 32 bits

- In the NEON unit, 8-bit operations are as fast as 32-bit operations
- So, don't promote shorter data

Avoid Conditions in Loops

- SIMD – Single instruction, multiple data
- Conditions (if, ?:, etc.) usually introduce conditional control-flow in the loop body
- Multiple control-flow operations -> Multiple PCs -> Multiple Instruction
  - Not allowed in SIMD
- Some NEON instructions allow elimination of control flow
  - Saturating math: VQADD, VQSUB, VQDMULH, etc.
  - Bitwise logic operations: VAND, VBIC, VEOR, VMVN, VORR, VORN
  - Bitwise select: VBIF, VBIT, VBSL
  - Comparison: VAC<cond>, VC<cond>, VTST
- Will the compiler generate these instructions?

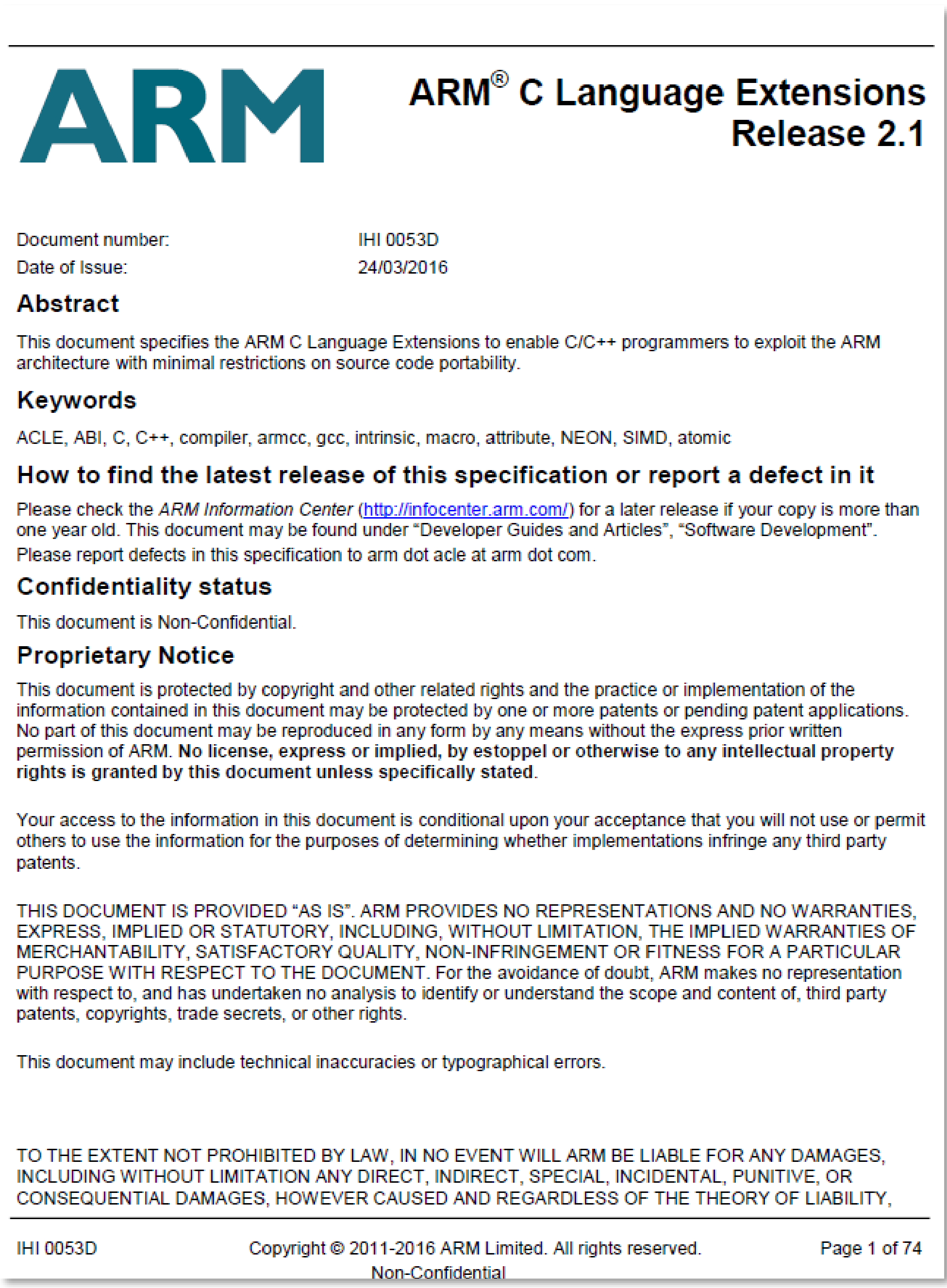


NC STATE UNIVERSITY

USING INTRINSICS AND ARM C-LANGUAGE EXTENSIONS

ARM C Language Extensions

- What are intrinsics?
  - Compiler keywords which specify architecture-specific operations: NEON and other instructions, support operations
  - May be implemented as functions, macros, other
- Where are the NEON intrinsics described?
  - NPG: Chapters 4 and 6
  - ARM: IHI0053D_acle_2_0.pdf, IHI0073A_arm_neon_intrinsics_ref.pdf
  - GCC: https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/ARM-C-Language-Extensions-_0028ACLE_0029.html
- How can we use them?
  - Header file
    - #include <arm_neon.h> for neon intrinsics
    - #include <arm_acle.h> for non-neon intrinsics
  - Makefile
    - –mfpu=neon



ARM® C Language Extensions
Release 2.1

Document number: IHI 0053D
Date of Issue: 24/03/2016

Abstract

This document specifies the ARM C Language Extensions to enable C/C++ programmers to exploit the ARM architecture with minimal restrictions on source code portability.

Keywords

ACLE, ABI, C, C++, compiler, armcc, gcc, intrinsic, macro, attribute, NEON, SIMD, atomic

How to find the latest release of this specification or report a defect in it

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than one year old. This document may be found under "Developer Guides and Articles", "Software Development".
Please report defects in this specification to arm dot acle at arm dot com.

Confidentiality status

This document is Non-Confidential.

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

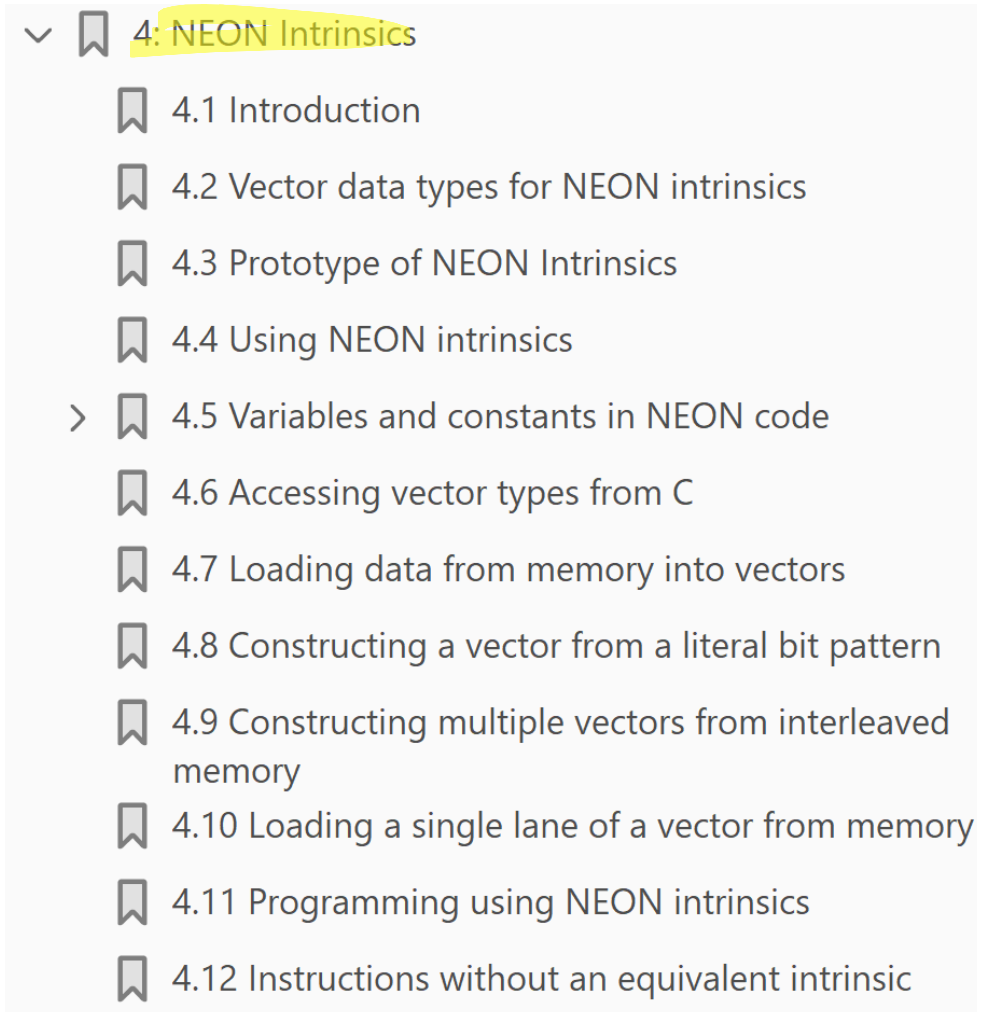
THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

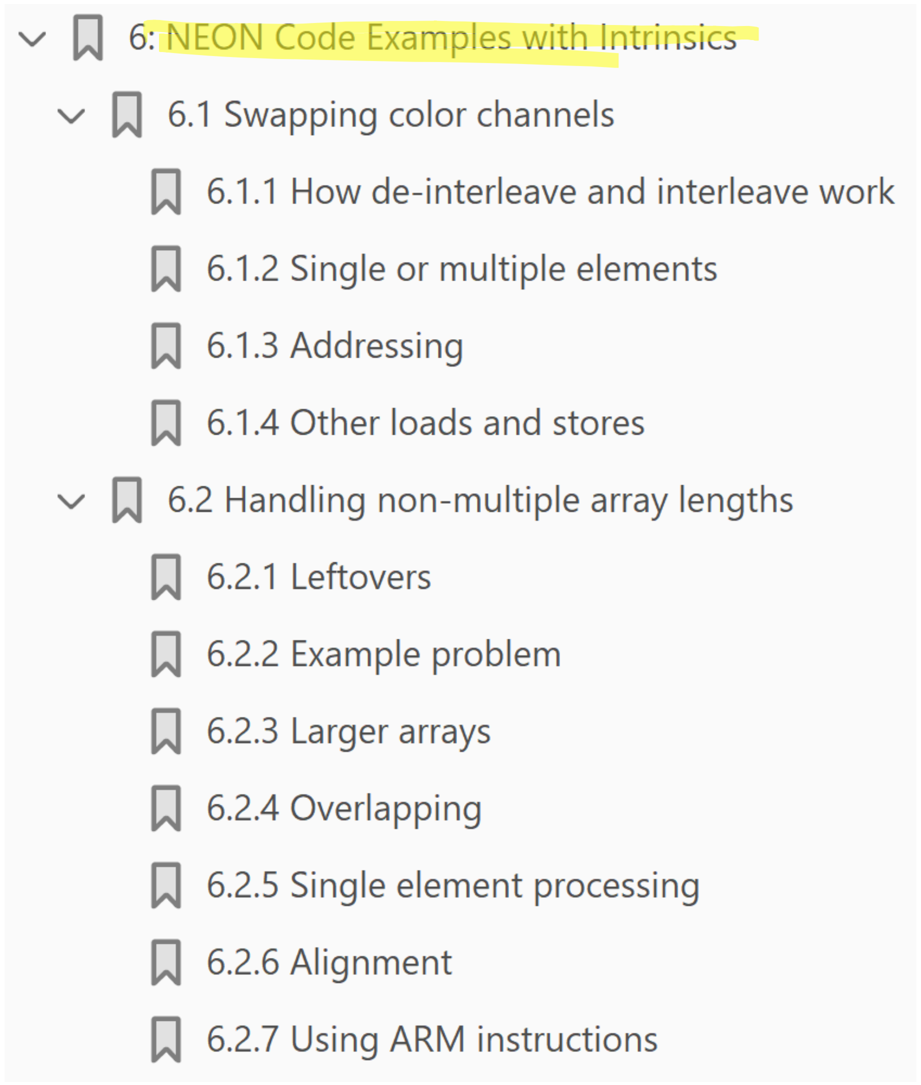
This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY,

IHI 0053D Copyright © 2011-2016 ARM Limited. All rights reserved. Page 1 of 74
Non-Confidential

NEON Programmer's Guide, Chapters 4 & 6

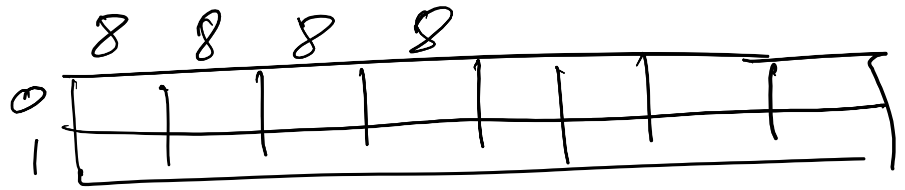




NC STATE UNIVERSITY

Data Types

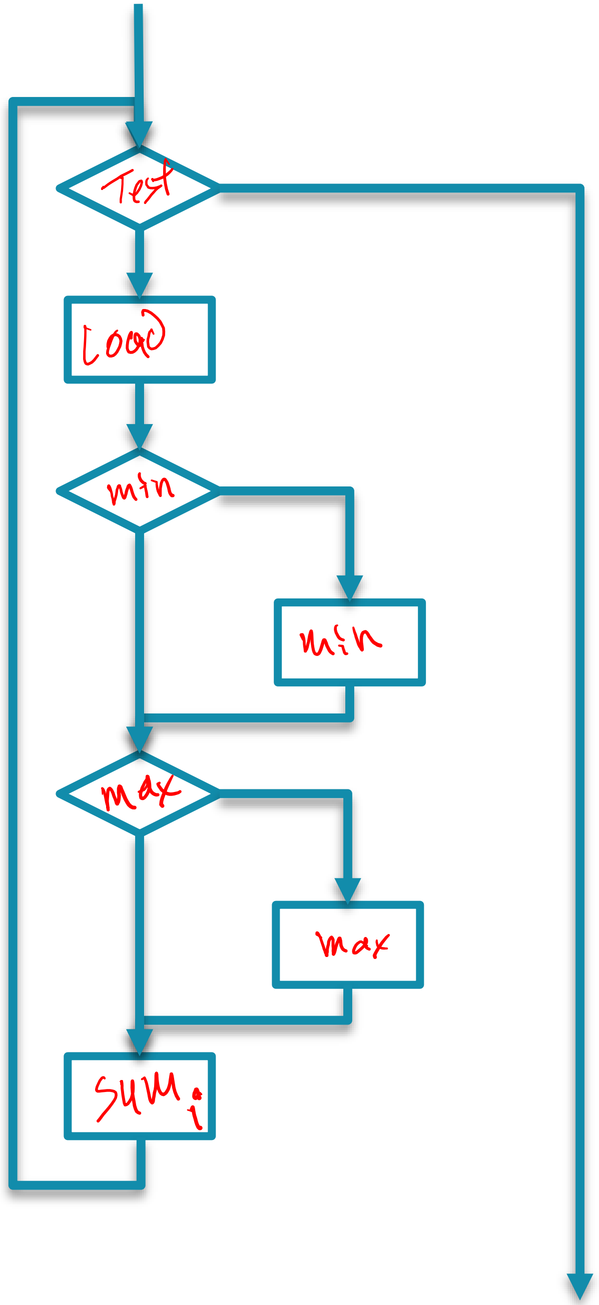
- Scalar data types
  - Based on standard types from <stdint.h>: int8_t, uint16_t, float32_t, float64_t
- Vector data types: base_typexvector_element_count_t
  - Lane type uses standard types from <stdint.h>
  - Multiple indicates vector element count
  - Examples: int8x8_t, float32x4_t
- Vector array types base_typexvector_element_countxarray_element_count_t
  - Based on vector data types, have multiples of 2, 3 or 4.
  - Examples: int8x8x2_t, int16x4x2_t



Example Program: Neon1

- In Speed/Vector/Neon1
- Find min, max and sum of values in array
- What will slow this code down the most?

```
float x[N_POINTS];
... main(...) {
 for (i=0; i<N_POINTS; i++) {
      if (x[i] < min_val)
        min_val = x[i];
      if (x[i] > max_val)
        max_val = x[i];
      sum_val += x[i];
    }
```



NC STATE UNIVERSITY

Manual Vectorization: Neon I

- Goal: operate on four float values at a time
- We will have to load the data, process it and reduce it
- Are there any vector min or max instructions available to eliminate control flow?
  - Examine NPG Chapter 3, Appendix C for instruction overviews

```
for (i=0; i<N_POINTS; i++) {
     if (x[i] < min_val)
       min_val = x[i];
     if (x[i] > max_val)
       max_val = x[i];
     sum_val += x[i];
}
```

NC STATE UNIVERSITY

Vector Max and Min Instructions

- **VMAX (Vector Maximum)** compares corresponding elements in two vectors, and writes the **larger** of them into the corresponding element in the destination vector.

- **VMIN (Vector Minimum)** compares corresponding elements in two vectors, and writes the **smaller** value into the corresponding element in the destination vector.

Syntax

```
VMAX{cond}.datatype Qd, Qn, Qm
VMAX{cond}.datatype Dd, Dn, Dm
VMIN{cond}.datatype Qd, Qn, Qm
VMIN{cond}.datatype Dd, Dn, Dm
```

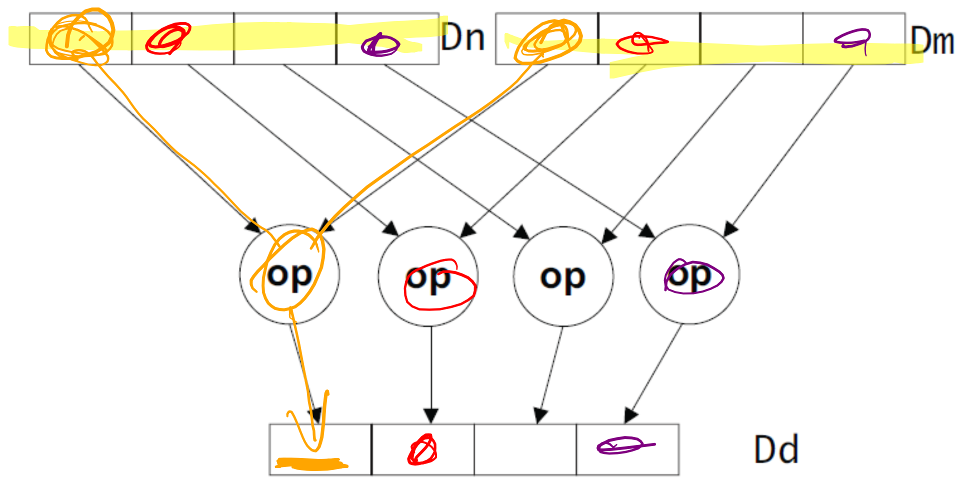
where:

cond is an optional conditional code.

datatype is one of S8, S16, S32, U8, U16, U32, or F32.

Qd, Qn, and Qm specify the destination, first operand and second operand registers for a quadword operation.

Dd, Dn, and Dm specify the destination, first operand and second operand registers for a doubleword operation.



Vector Pairwise Max and Min Instructions

- Pairwise: merging data lanes
- **VPMAX (Vector Pairwise Maximum)** compares adjacent pairs of elements in two vectors and writes the **larger** of each pair into the corresponding element in the destination vector.
- **VPMIN (Vector Pairwise Minimum)** compares adjacent pairs of elements in two vectors, and writes the **smaller** of each pair into the corresponding element in the destination vector.
- Operands and results must be **doubleword** vectors.

Syntax

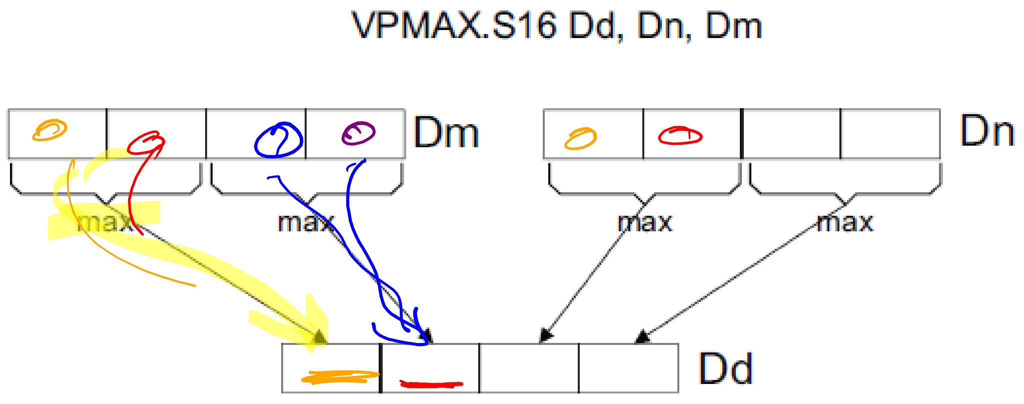
```
VPMAX{cond}.datatype Dd, Dn, Dm
VPMIN{cond}.datatype Dd, Dn, Dm
```

where:

cond is an optional conditional code.

datatype is one of S8, S16, S32, U8, U16, U32, or F32.

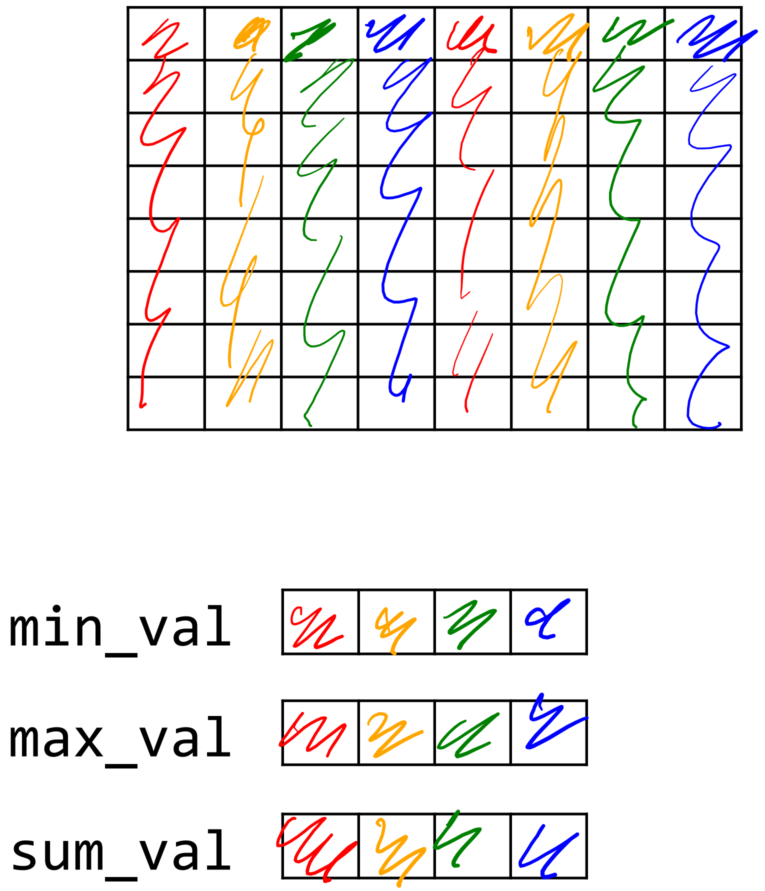
Dd, Dn, and Dm specify the destination, first operand and second operand registers for a doubleword operation.



NC STATE UNIVERSITY

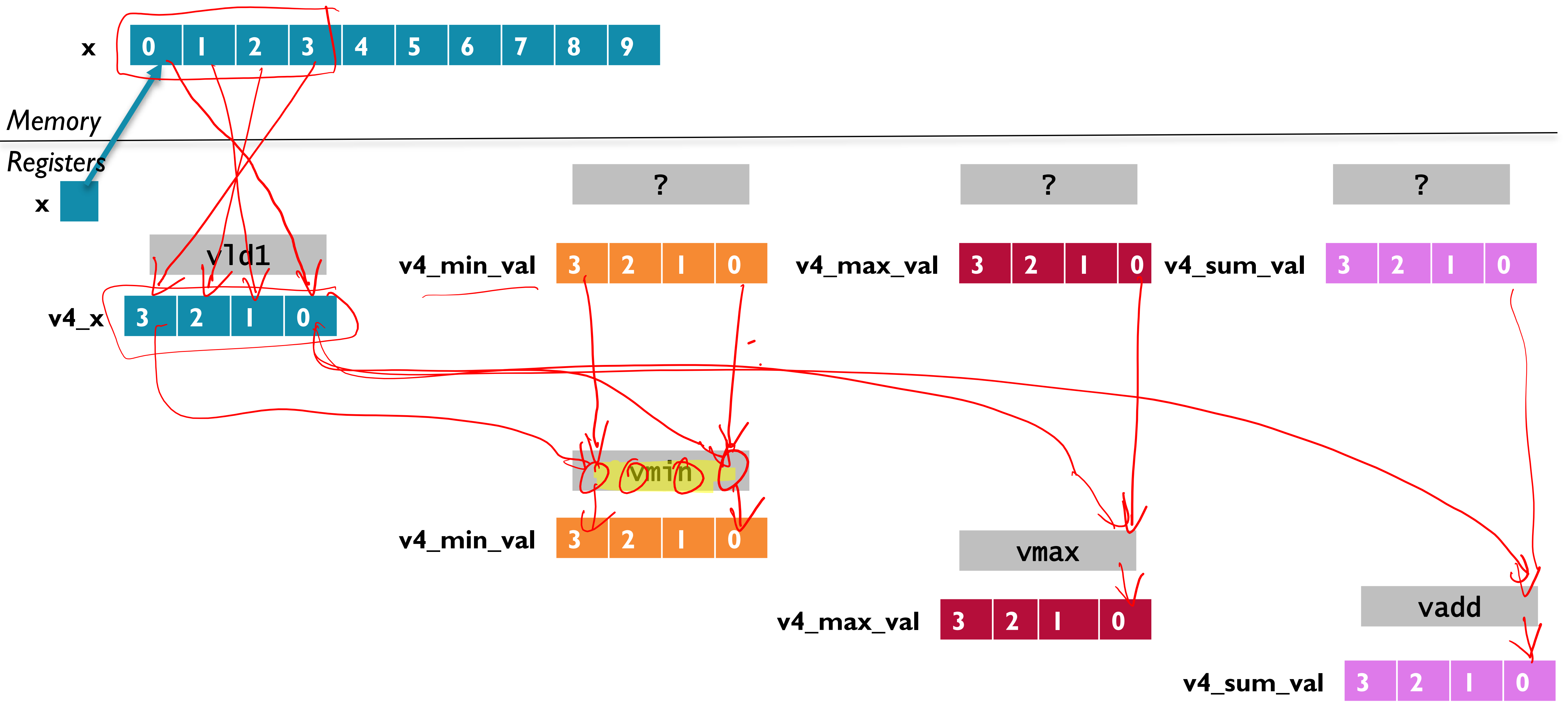
Program Outline

- Set up variables
- Vector processing loop
  - Load data elements from memory
  - Find each lane’s minimum
  - Find each lane’s maximum
  - Find each lane’s sum
- Vector reduction
  - Find minimum of all lane minima
  - Find maximum of all lane maxima
  - Find sum of all lane sums
- Clean-up processing for remaining iterations



NC STATE UNIVERSITY

Data Flow – Loop Body



Declare and Initialize Vector Variables

- See NPG Chapter 4
- Declare vector variables (v4_*) of type float32x4_t
- Initialize the vector variables. Two approaches:
  - Load scalar from memory and duplicate to all lanes
  - Load constant and duplicate to all lanes

```
int main (void) {
  struct timespec start, end, pre;
  long long diff;
  float sum_val = 0, min_val = 1e30, max_val = -1e30, el_time;
  int n, i;

  float32x4_t v4_x, v4_min_val, v4_max_val, v4_sum_val;
```

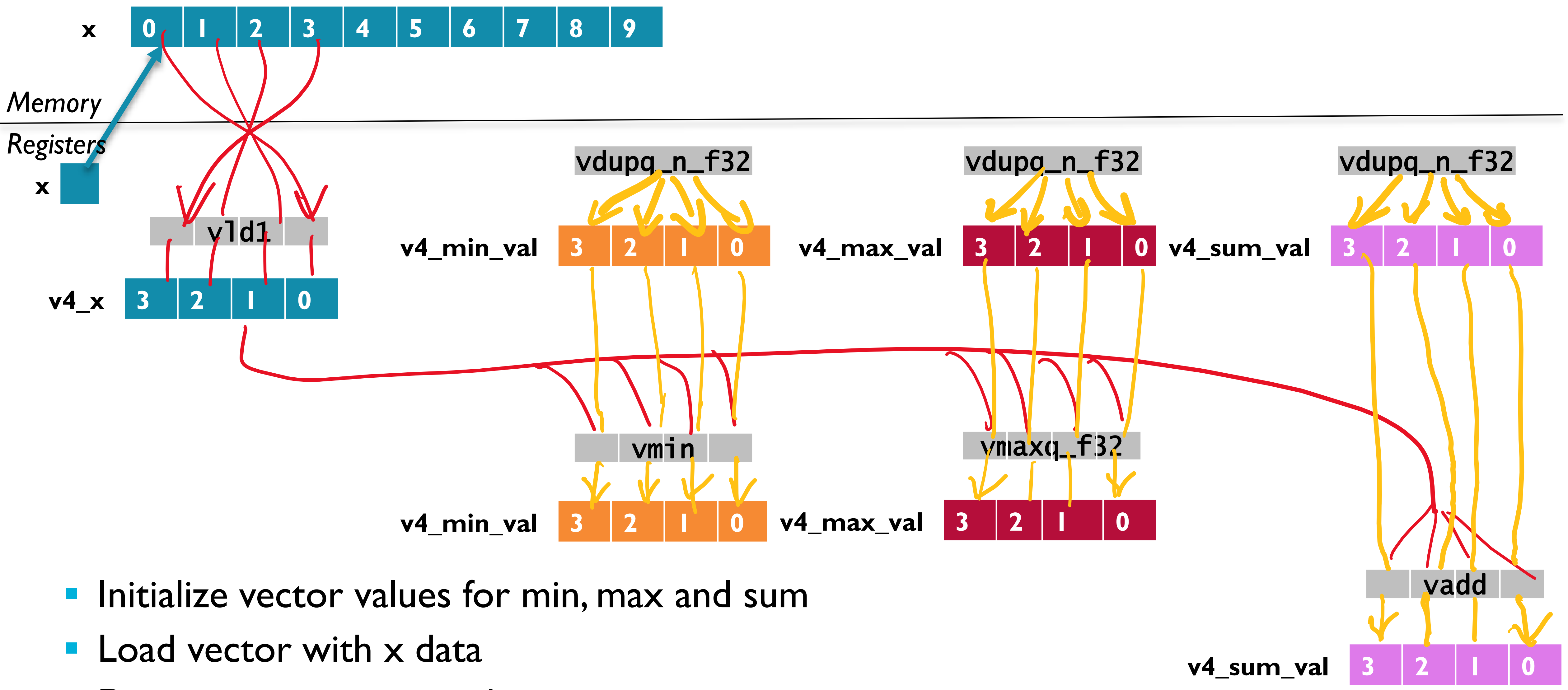
```
#if 0 // load values from memory
    v4_min_val = vld1q_dup_f32(&min_val); // q = quadword
    v4_max_val = vld1q_dup_f32(&max_val);
    v4_sum_val = vld1q_dup_f32(&sum_val);
#else // load values with constants
    v4_min_val = vdupq_n_f32(1e30);
    v4_max_val = vdupq_n_f32(-1e30);
    v4_sum_val = vdupq_n_f32(0);
#endif
```

Debug Support

```
void print_float32x4(float32x4_t v4) {
  int i;
  float v[4];
  float32x2_t v2;

  v2 = vget_low_f32(v4);
  v[0] = vget_lane_f32(v2, 0);
  v[1] = vget_lane_f32(v2, 1);
  v2 = vget_high_f32(v4);
  v[2] = vget_lane_f32(v2, 0);
  v[3] = vget_lane_f32(v2, 1);

  for (i=0; i<4; i++)
    printf("%f \t", v[i]);

}
```

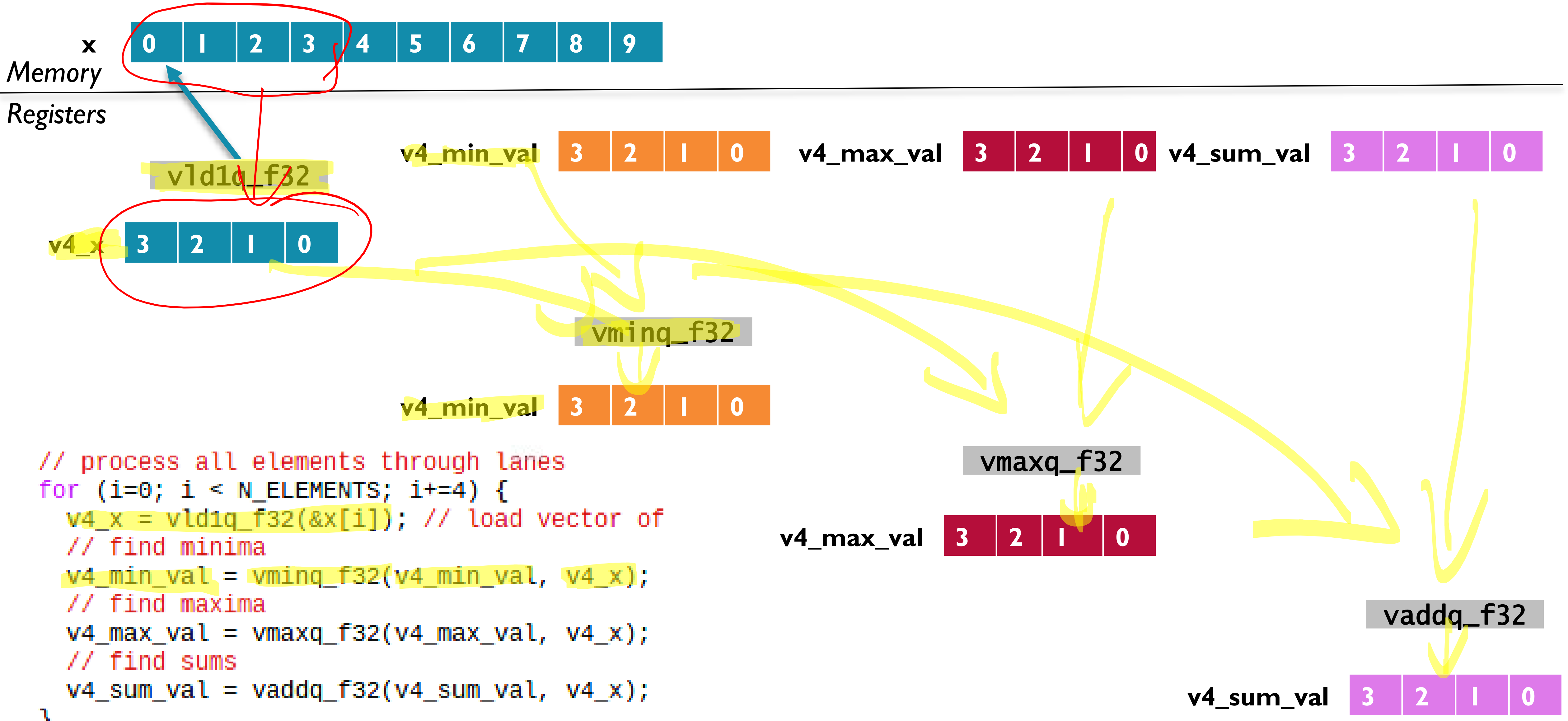
NC STATE UNIVERSITY

Data Flow Overview



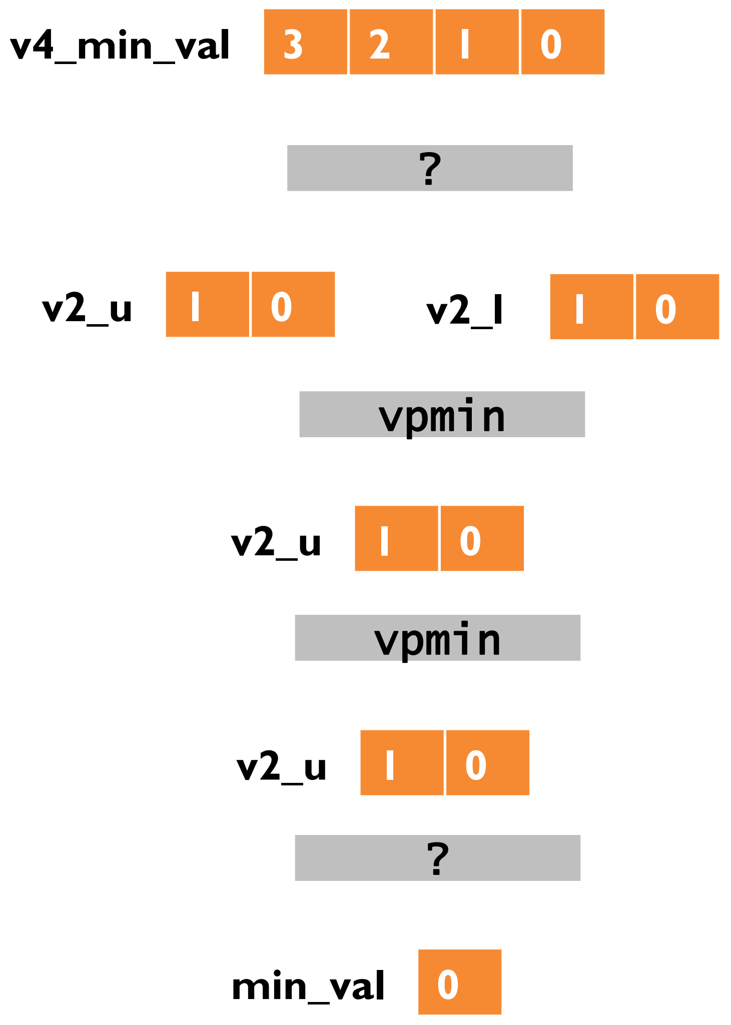
- Initialize vector values for min, max and sum
- Load vector with x data
- Determine min, max and sum

Code – Loop Body



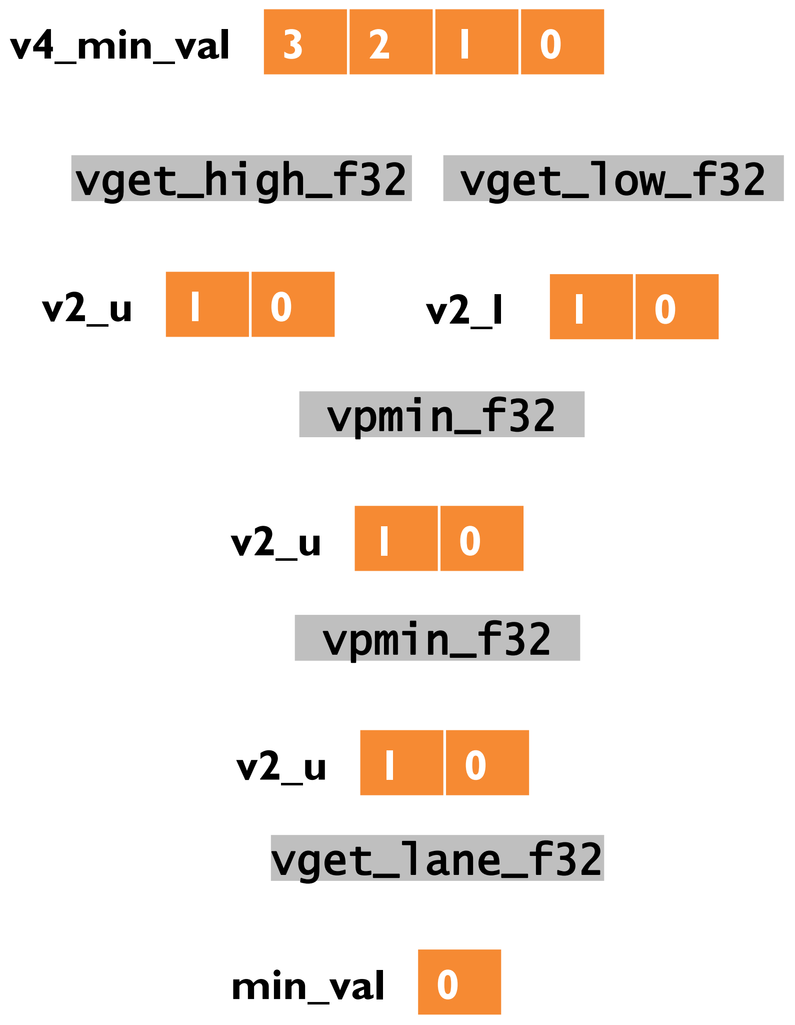
```
// process all elements through lanes
for (i=0; i < N_ELEMENTS; i+=4) {
  v4_x = vld1q_f32(&x[i]); // load vector of
  // find minima
  v4_min_val = vminq_f32(v4_min_val, v4_x);
  // find maxima
  v4_max_val = vmaxq_f32(v4_max_val, v4_x);
  // find sums
  v4_sum_val = vaddq_f32(v4_sum_val, v4_x);
}
```

Data Flow – Min (& Max) Reduction



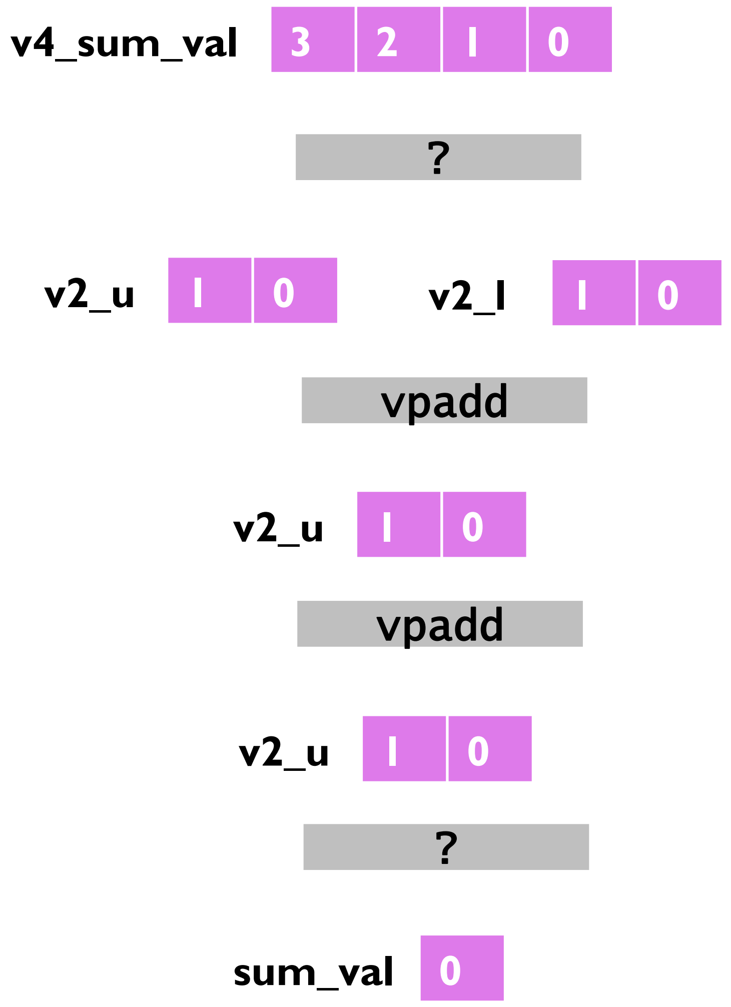
Need to use 2-element vectors since pairwise instructions work on D registers

Code – Min (& Max) Reduction

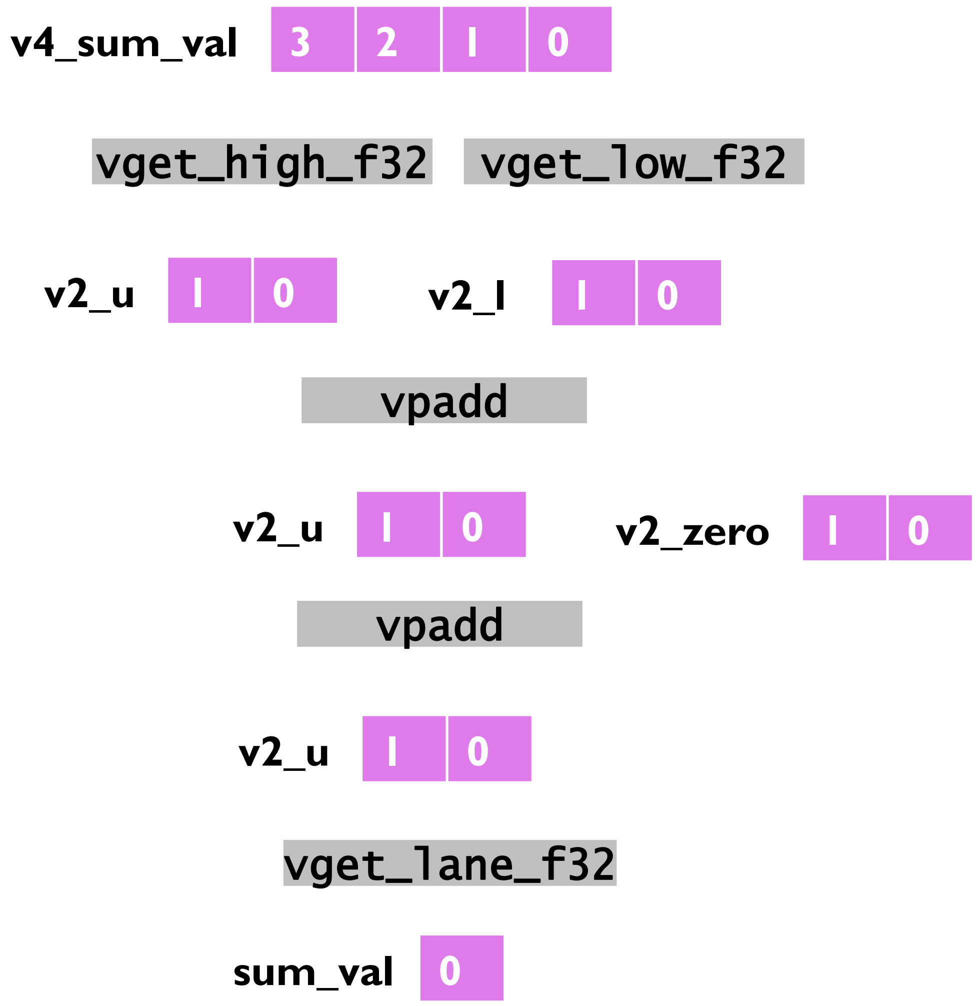


```
// Reduce lane results to single values
// min
float32x2_t v2_u, v2_l;
float32x2_t v2_zero = vdup_n_f32(0.0);

v2_u = vget_high_f32(v4_min_val);
v2_l = vget_low_f32(v4_min_val);
v2_u = vpmin_f32(v2_u, v2_l);
v2_u = vpmin_f32(v2_u, v2_u);
min_val = vget_lane_f32(v2_u, 0);

// max
v2_u = vget_high_f32(v4_max_val);
v2_l = vget_low_f32(v4_max_val);
v2_u = vpmax_f32(v2_u, v2_l);
v2_u = vpmax_f32(v2_u, v2_zero);
max_val = vget_lane_f32(v2_u, 0);
```

Data Flow – Sum Reduction

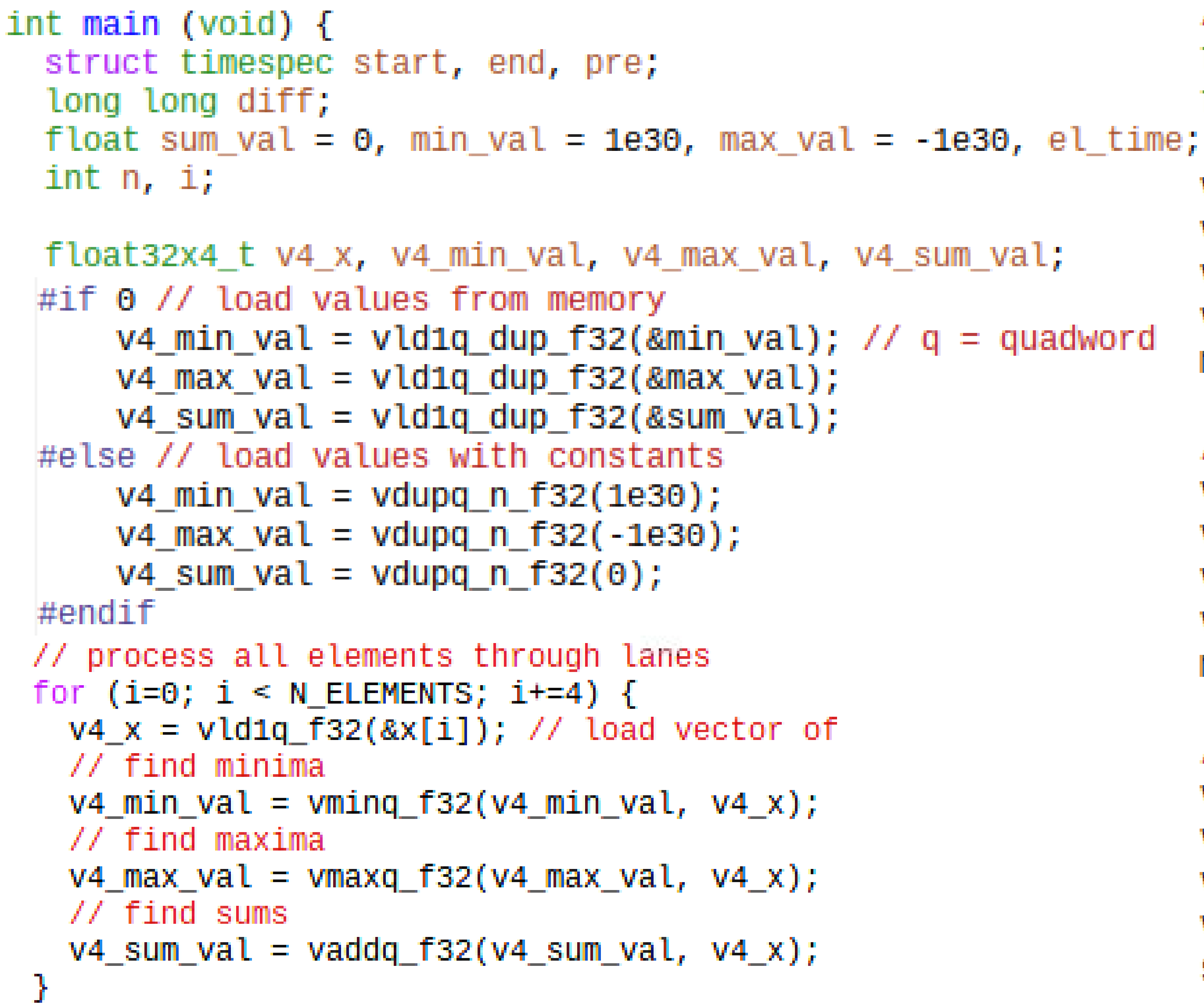


Code – Sum Reduction

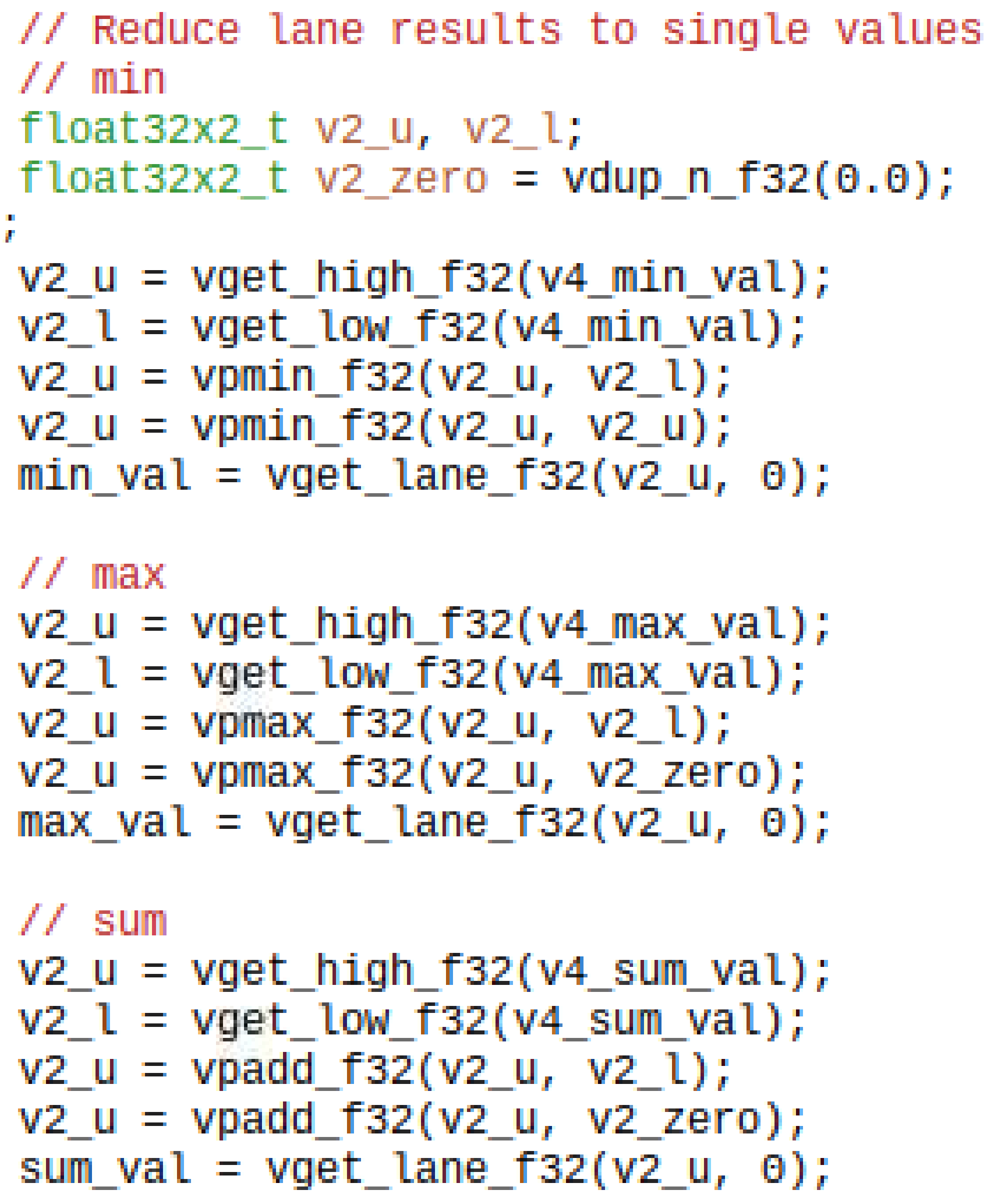


```
// sum
v2_u = vget_high_f32(v4_sum_val);
v2_l = vget_low_f32(v4_sum_val);
v2_u = vpadd_f32(v2_u, v2_l);
v2_u = vpadd_f32(v2_u, v2_zero);
sum_val = vget_lane_f32(v2_u, 0);
```

Resulting Code



```
int main (void) {
  struct timespec start, end, pre;
  long long diff;
  float sum_val = 0, min_val = 1e30, max_val = -1e30, el_time;
  int n, i;

  float32x4_t v4_x, v4_min_val, v4_max_val, v4_sum_val;
 #if 0 // load values from memory
     v4_min_val = vld1q_dup_f32(&min_val); // q = quadword
     v4_max_val = vld1q_dup_f32(&max_val);
     v4_sum_val = vld1q_dup_f32(&sum_val);
 #else // load values with constants
     v4_min_val = vdupq_n_f32(1e30);
     v4_max_val = vdupq_n_f32(-1e30);
     v4_sum_val = vdupq_n_f32(0);
 #endif
 // process all elements through lanes
 for (i=0; i < N_ELEMENTS; i+=4) {
   v4_x = vld1q_f32(&x[i]); // load vector of
   // find minima
   v4_min_val = vminq_f32(v4_min_val, v4_x);
   // find maxima
   v4_max_val = vmaxq_f32(v4_max_val, v4_x);
   // find sums
   v4_sum_val = vaddq_f32(v4_sum_val, v4_x);
 }
```



```
// Reduce lane results to single values
// min
float32x2_t v2_u, v2_l;
float32x2_t v2_zero = vdup_n_f32(0.0);

v2_u = vget_high_f32(v4_min_val);
v2_l = vget_low_f32(v4_min_val);
v2_u = vpmin_f32(v2_u, v2_l);
v2_u = vpmin_f32(v2_u, v2_u);
min_val = vget_lane_f32(v2_u, 0);

// max
v2_u = vget_high_f32(v4_max_val);
v2_l = vget_low_f32(v4_max_val);
v2_u = vpmax_f32(v2_u, v2_l);
v2_u = vpmax_f32(v2_u, v2_zero);
max_val = vget_lane_f32(v2_u, 0);

// sum
v2_u = vget_high_f32(v4_sum_val);
v2_l = vget_low_f32(v4_sum_val);
v2_u = vpadd_f32(v2_u, v2_l);
v2_u = vpadd_f32(v2_u, v2_zero);
sum_val = vget_lane_f32(v2_u, 0);
```

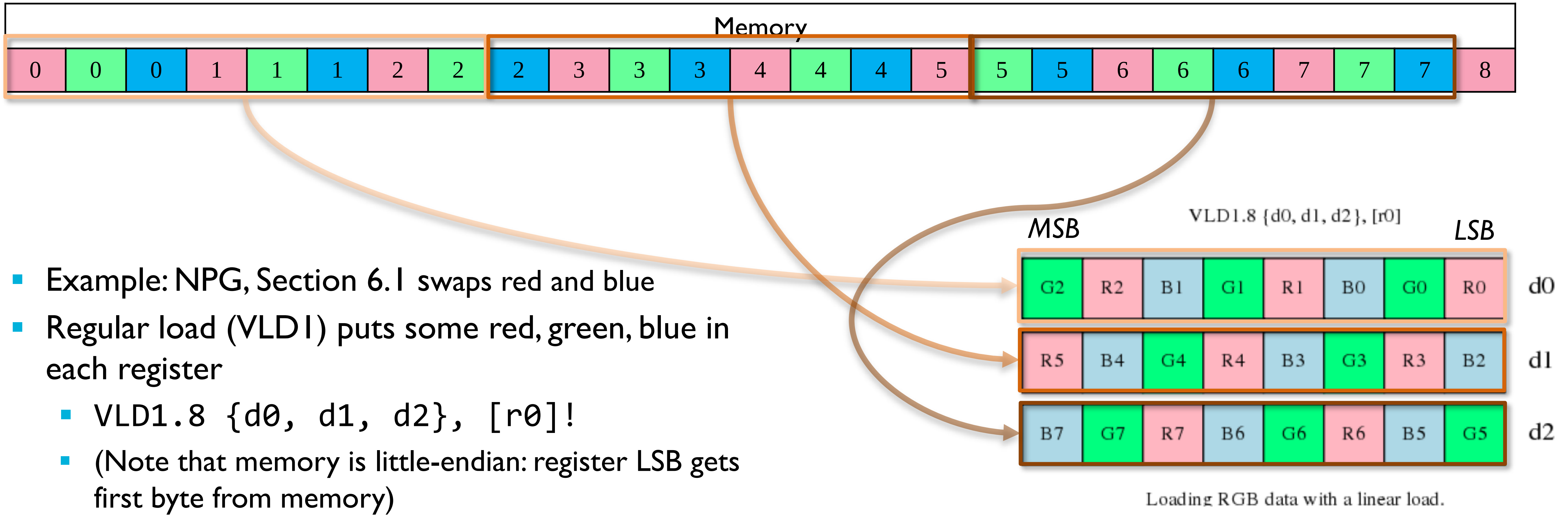
Can the Compiler Vectorize Neon1?

- Try it out
  - -O1?
  - -O2?
  - -O3?
  - -Ofast?

Interleaving and De-Interleaving

NC STATE UNIVERSITY

Memory Layout May Not Match Vector Layout



Loading RGB data with a linear load.

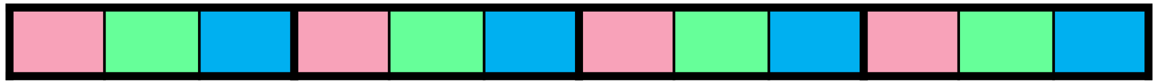
- Example: NPG, Section 6.1 swaps red and blue
- Regular load (VLD1) puts some red, green, blue in each register
  - `VLD1.8 {d0, d1, d2}, [r0]!`
  - (Note that memory is little-endian: register LSB gets first byte from memory)
- Could rewrite code so data in memory is a structure of arrays instead:

```
struct {
   uint8_t Red[N], Green[N], Blue[N];
} image;
```

Arrays and Structures

- **Array of structures**

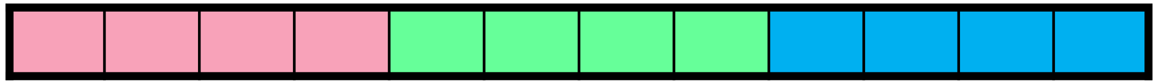
```
struct {
   uint8_t Red, Green, Blue;
} image[N];
```



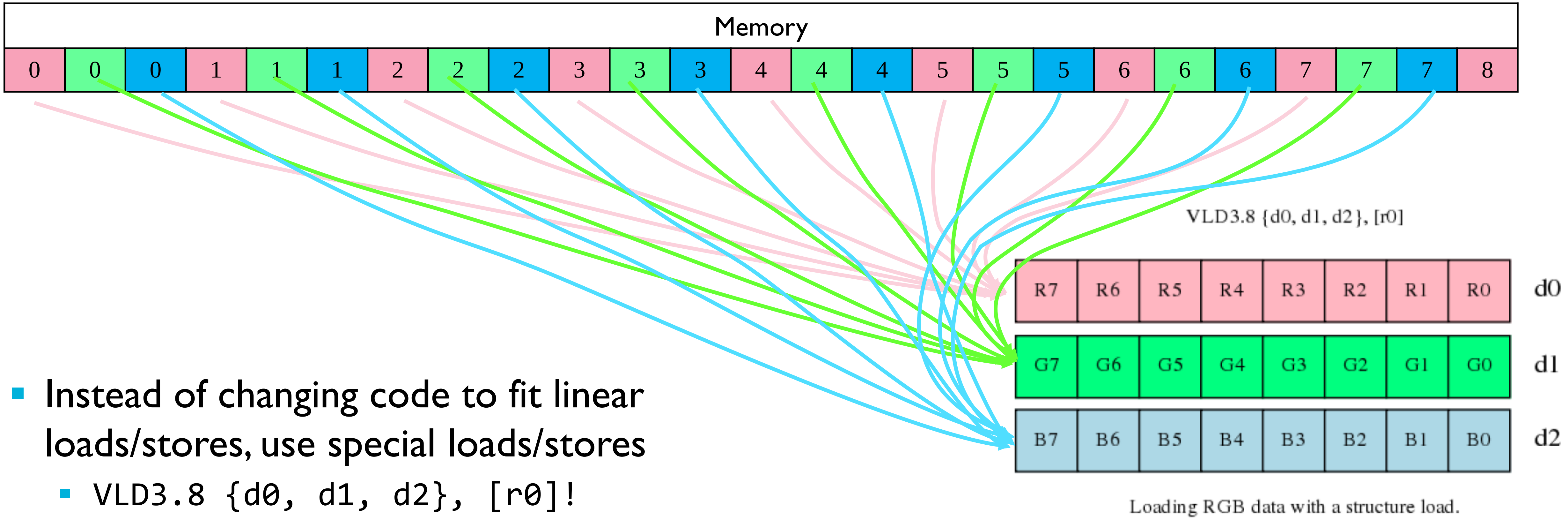
- Could rewrite code to rearrange data in memory into a **structure of arrays**:

```
struct {
   uint8_t Red[N], Green[N], Blue[N];
} image;
```

  - Is better fit for normal (linear) vector loads



"Structure Load" De-Interleaves From Memory Into Register



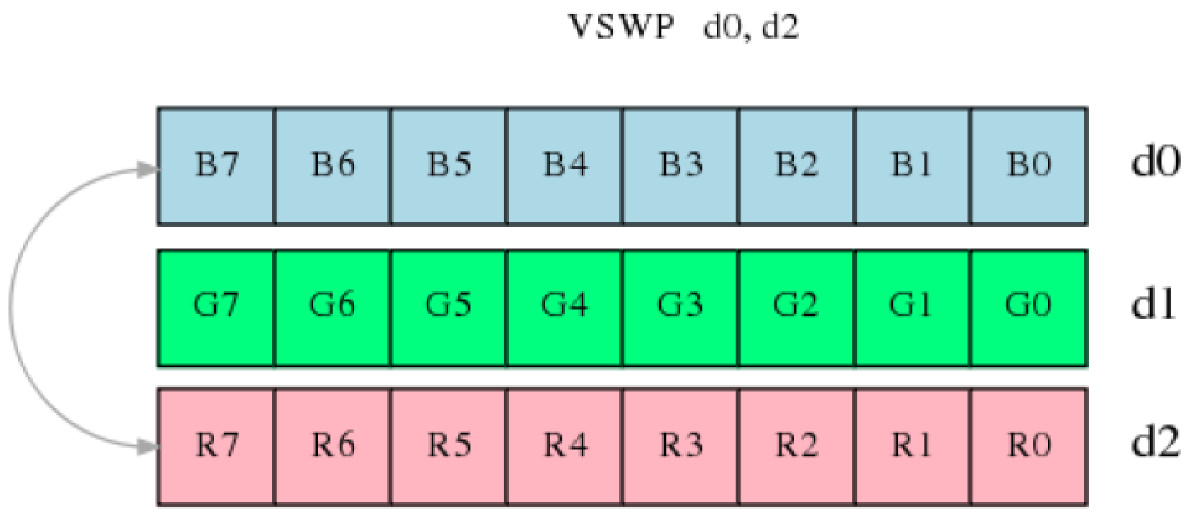
Loading RGB data with a structure load.

- Instead of changing code to fit linear loads/stores, use special loads/stores
  - `VLD3.8 {d0, d1, d2}, [r0]!`
  - Structure load (VLD**n**) de-interleaves memory into **n** separate registers
- Instructions: NPG, page C-63

Swap Registers

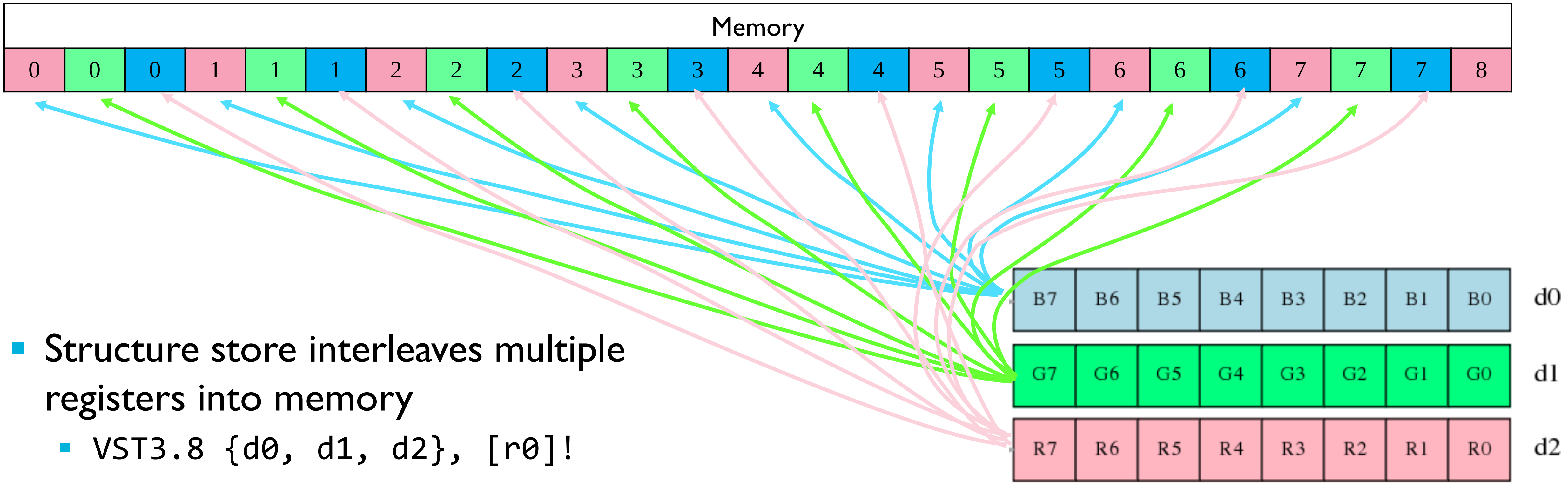
- Now can swap red and blue easily

```
VSWP d0, d2
```



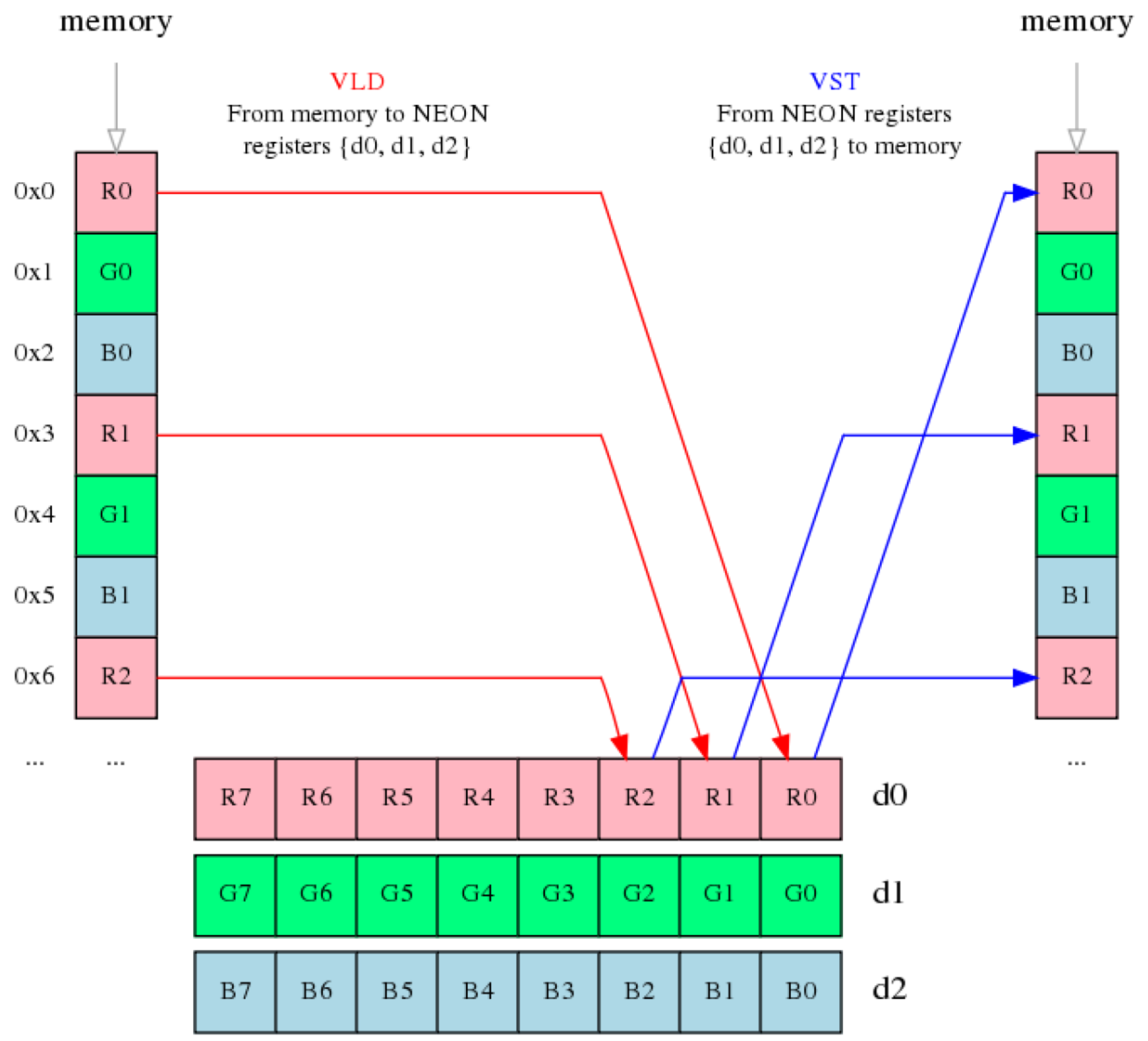
Swapping the contents of registers d0 and d2.

"Structure Store" Interleaves From Register Into Memory



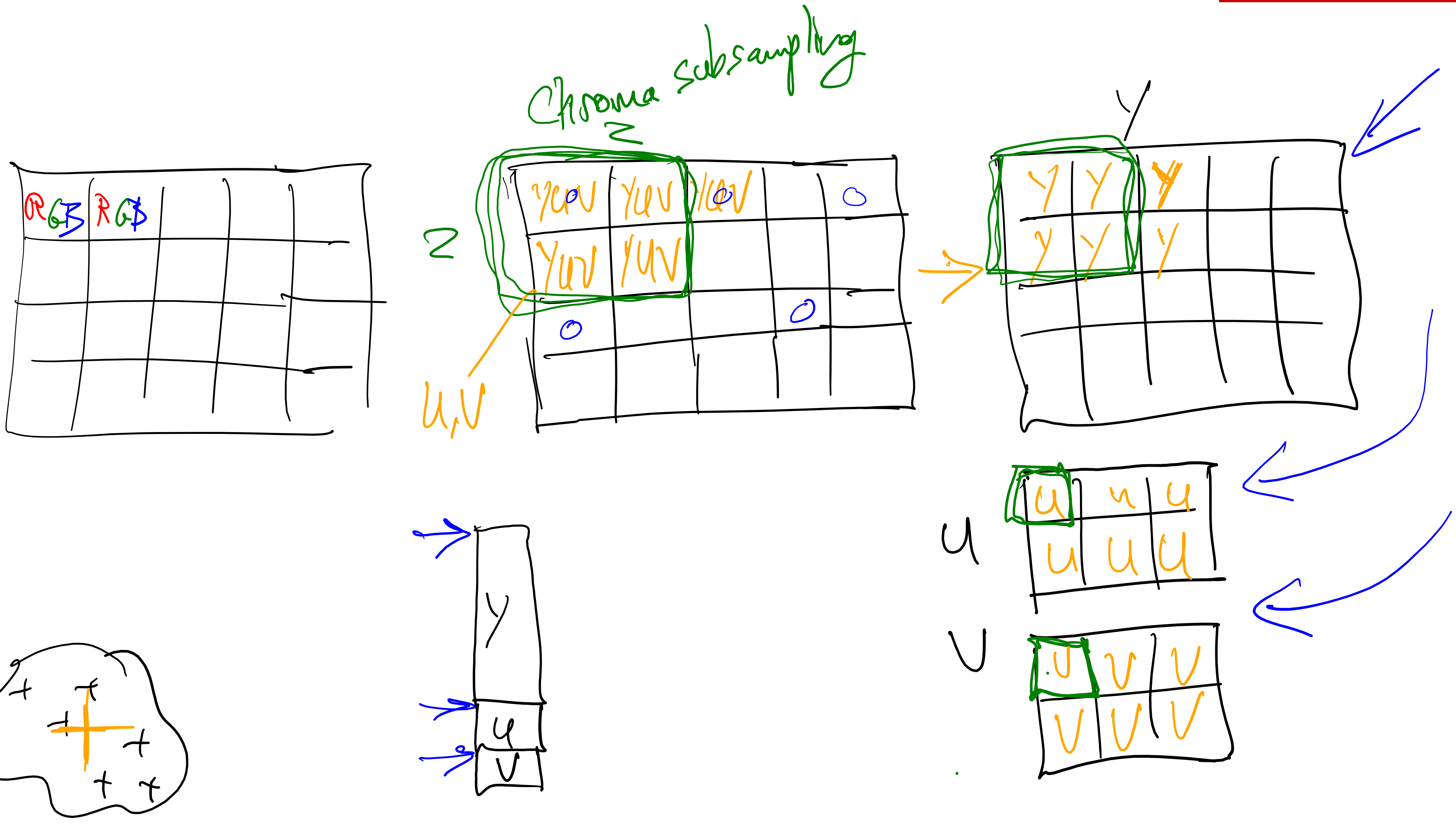
- Structure store interleaves multiple registers into memory
  - `VST3.8 {d0, d1, d2}, [r0]!`

Big Picture



NEON structure loads and stores.

- Have support for 2, 3 and 4 element structures
- How can it work?
  - Wide interfaces between NEON registers and memory
  - L1 Data Cache
    - 128 bit interface


Chroma subsampling
RGB RGB
2
2
YUV YUV YUV
YUV YUV
U,V
Y
Y Y Y
Y Y Y
U
U U U
U U U
V
V V V
V V V
Y
U
V