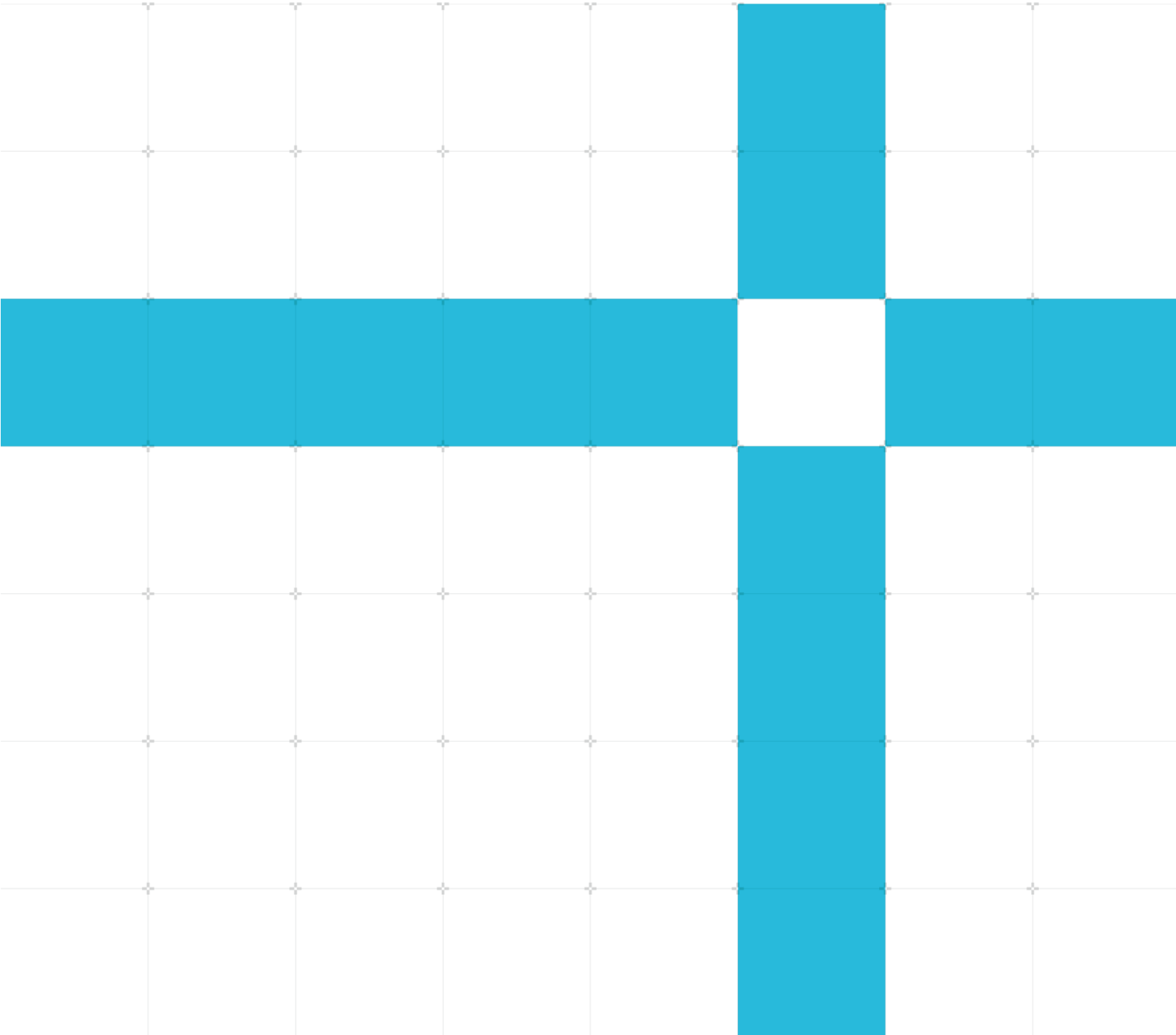
arm

Neoverse N1

Accelerating DSP functions with dot product instructions

Non-Confidential

Issue 01
102651



Neoverse N1

Accelerating DSP functions with dot product instructions

Release information

Document history

Issue	Date	Confidentiality	Change
01	December 14, 2021	Non-confidential	Initial draft

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

1 Overview

In this guide, learn about the Armv8.4-A dot product instructions, which are available in Cortex-A75, Cortex-A55, and Neoverse N1 and later. This guide introduces the following three use cases for the dot product instructions:

- Convolution
- Averaging
- Taking the Sum of Absolute Differences (SAD)

These operations are used to improve the performance of the libvpx implementation of VP9. At the end of this guide, you will understand the use cases that the dot product instructions can enable and how to apply them to digital signal processing code of your own.

1.1 Before you begin

This guide assumes you are familiar with the Cortex-A processors and Neoverse. If you are not familiar with the Cortex-A architecture, see the Arm Cortex-A series processors page. To learn about Neoverse, see the Neoverse site.

2 What are dot product instructions?

Arm introduced the SDOT (Signed Dot Product) and UDOT (Unsigned Dot Product) instructions in the 2017 extensions to the Arm Architecture, known as Armv8.4-A.

These vector instructions operate on 32-bit elements within 64-bit or 128-bit vectors in the Neon instruction set or within scalable vectors in the Scalable Vector Extensions (SVE2) instruction set. In these 32-bit elements are four 8-bit elements. Each 8-bit element in each 32-bit element of the first vector is multiplied by the corresponding 8-bit element in the second vector, creating four sets of four products. Each group of four products are added to create a 32-bit sum, and this sum is accumulated into the 32-bit element of the destination vector. Conceptually, this is the vector inner, dot, or scalar product. In this guide, we use the term dot product to match the instruction name. The following diagram shows how the vector instructions operate:

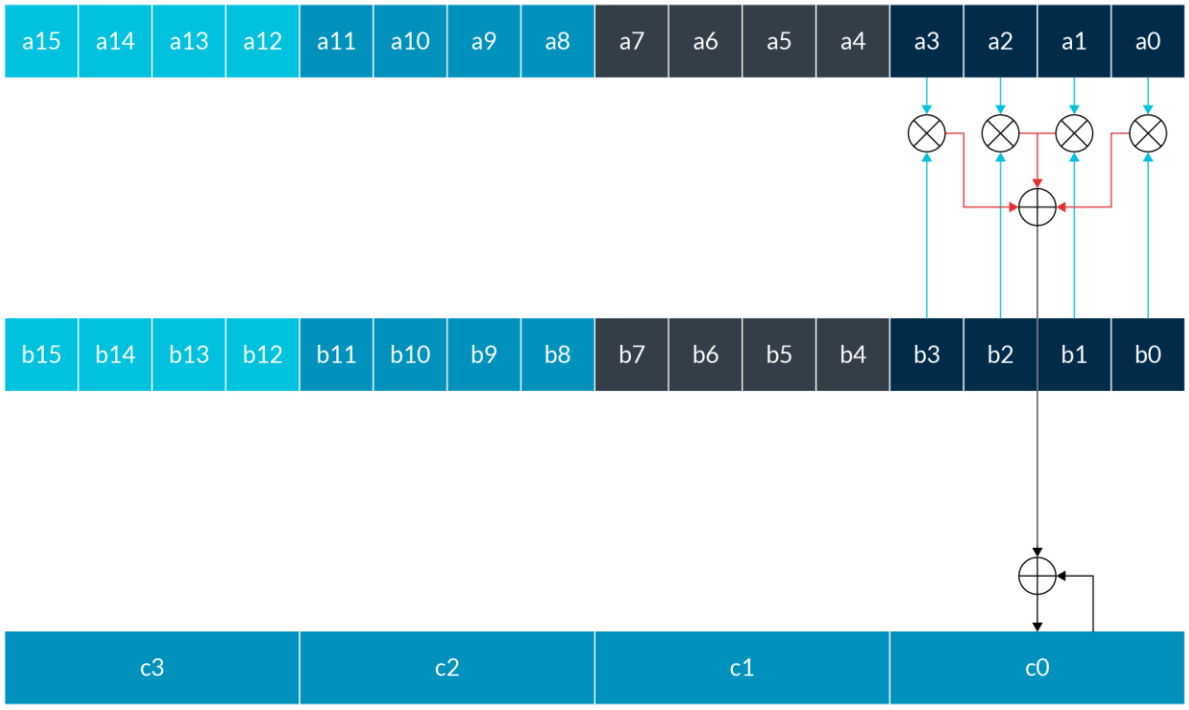


Figure 1: Vector instructions

For example, the operation performed on the first set of elements is:

c0 = c0 + ((a0 * b0) + (a1 * b1) + (a2 * b2) + (a3 * b3))

The dot product instructions provide access to many multiply and accumulate operations every cycle. Processors such as the Arm Cortex-X2 and Arm Neoverse V1 can compute four dot product instructions in parallel. This computation allows us to multiply four 8-bit elements in four 32-bit subvectors across four 128-bit parallel operations every cycle. This works out to sixty four 8-bit multiply and (partial) accumulate operations per cycle. The following diagram shows the SDOT operations per cycle:

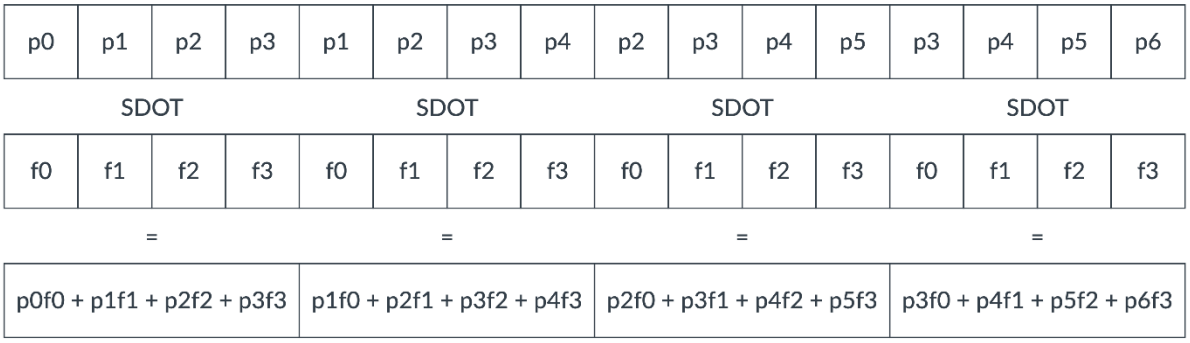


Figure 2: SDOT operations

3 Calculating a one-dimensional image convolution

In a convolution, we perform a filter function over values either side of our current element and write back the result. For example, one common filter function is a weighted average of pixel values. To calculate this average, take a multiplication by a set of constants and sum them to a single value. With the appropriate data layout, this is a dot product between the input elements and the filter values. For example, to compute the weighted average of eight 8-bit values, use two rounds of the dot product instruction as shown in the following code:

```
#include "arm_neon.h"

uint32x4_t weighted_average (uint8x16_t values_low,
                             uint8x16_t values_high,
                             uint8x8_t weights) {
    uint32x4_t result = vdupq_n_u32 (0);
    /* Low values multiplied by the first four weights.  */
    result = vdotq_lane_u32 (result, values_low, weights, 0);
    /* Accumulate with high values multiplied by the next four weights.  */
    result = vdotq_lane_u32 (result, values_high, weights, 1);
    return vshrq_n_u32 (result, 3);
}
```

This code generates the following instructions with GCC 11.1:

```
weighted_average:
        movi    v3.2d, 0
        udot    v3.4s, v0.16b, v2.4b[0]
        udot    v3.4s, v1.16b, v2.4b[1]
        ushr    v0.4s, v3.4s, 3
        ret
```

For more information about this code, see Godbolt. To get maximum parallelism out of the DOT instruction, we compute four output lanes at one time. Because we apply this filter, we can create an appropriate data layout by loading sixteen values at a time using vld1q_u8, and then use the TBL instructions to rearrange data. This example is shown in the following diagram:

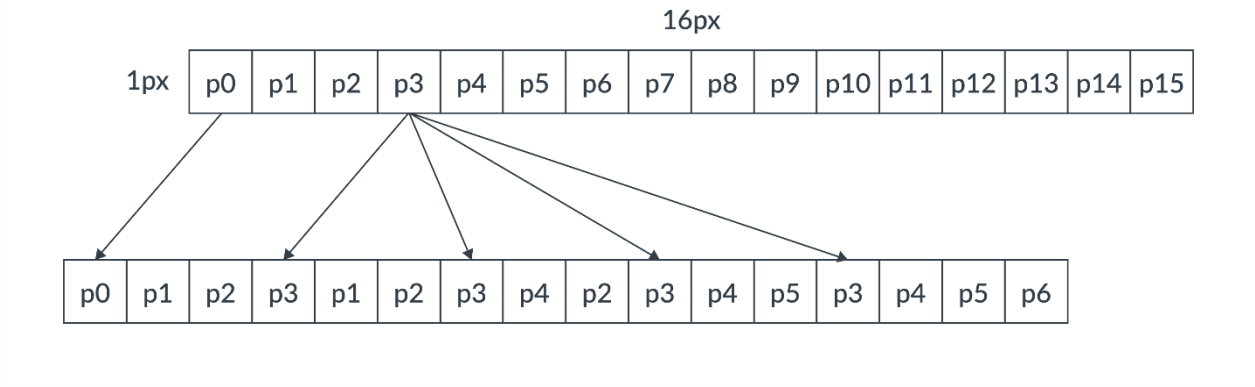


Figure 3: Data layout example

Now we can create our first four output values, as shown in the following diagram:

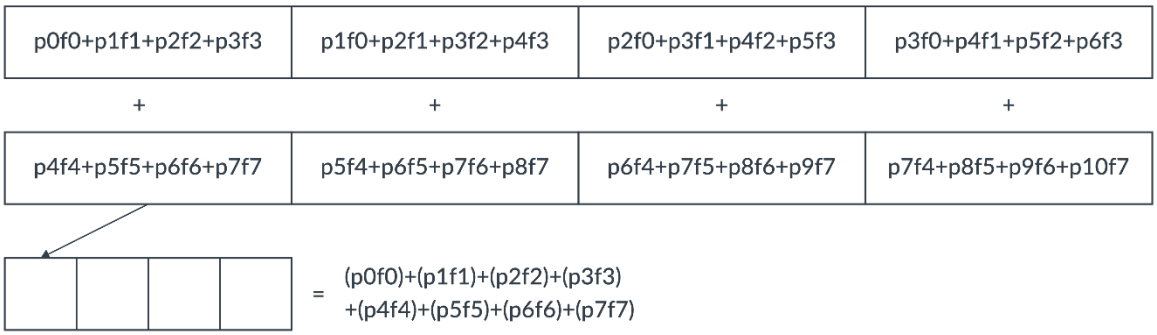


Figure 4: Output value example

Notice that the `values_high` calculated can be used as the `values_low` value for the next four pixels. We use one more TBL instruction to generate the next `values_high` and complete our calculation. We then take the eight 32-bit results and reduce them back to eight 8-bit output values, using the following methods:

- vqmovn_u32 to narrow a 32-bit value to a 16-bit value with saturation.
- vcombine_u16 to pack two vectors of four 16-bit values and create one vector of eight 8-bit values.
- vqshrn_n_u16 to saturate, narrow, and shift a result.

4 Calculating an average

The average over a large array is a weighted average where all weights are set to one. We can use this same strategy multiplying by a vector of one to perform widening additions in parallel. Because these widening additions perform 16 parallel partial sums, this can be quicker than using pairs of other Armv8-A instructions like `UADDL` and `UADDL2`.

This calculation is shown in the following code:

```
#include "arm_neon.h"
#define N 4096
// 16 elements in a vector
#define STRIDE (16)

unsigned int average (uint8_t *in) {
    uint32x4_t sum = vmovq_n_u32 (0);
    uint8x16_t ones = vmovq_n_u8 (1);
    for (int i = 0; i < N; i += STRIDE) {
        sum = vdotq_u32 (sum, vld1q_u8 (in), ones);
        in += STRIDE;
    };
    return vaddvq_u32 (sum) / N;
}
```

5 Calculating the SAD

In a Sum of Absolute Differences (SAD) computation, we add together the absolute difference of each item in two arrays and return the result. In C code this would look like the following snippet:

```
unsigned int sad (uint8_t *x, uint8_t *y) {
    unsigned int result;
    for (int i = 0; i < N; i++)
        result += abs (x[i] - y[i]);
    return result;
}
```

While Neon in Armv8.0-A contains instructions to accelerate the calculation of SAD, these instructions operate on each lane and must use a wider type for intermediate results. This means that we need more instructions on each loop iteration. The dot product instructions allow us to do this in one step. It is important to note that multiplication by 1 returns the same value. These two code generation strategies are shown in the following code generated with GCC 11.1:

Without dot product:

```
// During the loop
  uabdl2  v0.8h, v1.16b, v2.16b
  uabal   v0.8h, v1.8b, v2.8b
  uadalp  v3.4s, v0.8h

// After the loop
  addv    s3, v3.4s
```

With dot product:

```
// Before the loop
  movi    v3.16b, 0x1

// During the loop
   abd    v0.16b, v0.16b, v1.16b
   udot    v2.4s, v0.16b, v3.16b

// After the loop
   addv    s2, v2.4s
```

For more information about this code, see Godbolt. Not only does this optimization reduce the number of instructions executed within the loop body, but it can also avoid resource utilization differences between the `UABDL2`, `UABAL`, and `UDALP` instructions. This optimization allows better throughput of the summation operations and increases overall performance. Further benefits can come from unrolling the loop multiple times, making better use of available hardware parallelism. For example, we can rewrite this example using Neon intrinsics as shown in the following snippet:

```
#include "arm_neon.h"
```

```
#define N 4096
/* Unroll 4x, calculate 16 items per vector.  */
#define STRIDE (4 * 16)
unsigned int sad_unrolled (uint8_t *x, uint8_t *y) {
    uint32x4_t p0, p1, p2, p3;
    uint8x16_t x0, x1, x2, x3;
    uint8x16_t y0, y1, y2, y3;
    p0 = p1 = p2 = p3 = vmovq_n_u32 (0);
    uint8x16_t ones = vmovq_n_u8 (1);
    for (int i = 0; i < N; i += STRIDE) {
         x0 = vld1q_u8 (x + 0 );
         x1 = vld1q_u8 (x + 16);
         x2 = vld1q_u8 (x + 32);
         x3 = vld1q_u8 (x + 48);
         y0 = vld1q_u8 (y + 0 );
         y1 = vld1q_u8 (y + 16);
         y2 = vld1q_u8 (y + 32);
         y3 = vld1q_u8 (y + 48);
        p0 = vdotq_u32 (p0, vabdq_u8 (x0, y0), ones);
        p1 = vdotq_u32 (p1, vabdq_u8 (x1, y1), ones);
        p2 = vdotq_u32 (p2, vabdq_u8 (x2, y2), ones);
        p3 = vdotq_u32 (p3, vabdq_u8 (x3, y3), ones);
        x += STRIDE;
        y += STRIDE;
    };
    return vaddvq_u32 (vaddq_u32 (vaddq_u32 (p0, p1), vaddq_u32 (p2, p3)));
```

This approach of unrolling to break dependency accumulation chains can provide benefits across a range of Neon instructions, enabling more instruction level parallelism on the highest performance cores. This optimization is done by hand because for saturating operations and floating-point operations, the order of operations impacts results. A compiler cannot know whether it is safe to accumulate in a different order.

6 Use case: improving VP9 performance

Libvpx is an open-source library that provides reference implementations of the VP8 and VP9 video codecs. It is available as part of the WebM project and you can find the code on Google Git. To accelerate VP9 performance on the latest cores, some of the core functions of the VP9 encoder use the dot product instructions.

The standard Linux performance analysis tools `perf record` and `perf report` are used to understand where the encoder spends time. The experiments were completed using the Neoverse N1 SDP platform with a Clang 12 compiler, as shown in the following code:

```
$ perf record ./vpxenc --codec=vp9 --height=1080 --width=1920 --fps=25/1 --limit=20
$ perf report
    14.60%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_convolve8_horiz_neon
     7.43%  vpxenc-12a14913  vpxenc-12a149139  [.] vp9_optimize_b
     7.00%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_convolve8_vert_neon
     4.60%  vpxenc-12a14913  vpxenc-12a149139  [.] vp9_diamond_search_sad_c
     4.21%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_sad16x16x4d_neon
     3.19%  vpxenc-12a14913  vpxenc-12a149139  [.] rd_pick_best_sub8x8_mode
     2.90%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_sad32x32x4d_neon
     2.76%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_quantize_b_neon
     2.24%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_quantize_b_32x32_neon
     1.53%  vpxenc-12a14913  vpxenc-12a149139  [.] vpx_variance32x32_neon
```

From the names of the functions in the above report, we can see that there are already optimized paths in the code that use the Advanced SIMD architecture. Looking in more detail, we can identify several target functions for further optimization:

- `vpx_convolve8_horiz_neon` `vpx_convolve8_vert_neon`

These functions are optimized using the approach described in Calculating a one-dimensional image convolution. The functions use the dot product instructions to increase the available multiply and accumulate throughput available to us.

The following patches to VP9 perform this optimization:

- Implement horizontal convolution using Neon SDOT instruction
- Implement vertical convolution using Neon SDOT instruction
- Merge transpose and permute in Neon SDOT vertical convolution

The following patches optimize the averaging versions of these convolutions:

- Implement vpx_convolve8_avg_vert_neon using SDOT instruction
- Implement vpx_convolve8_avg_vert_neon using SDOT instruction
- `vpx_sad16x16x4d_neon`, `vpx_sad32x32x4d_neon`

We optimized these functions using the approach described in Calculating the SAD.

The following patch implements this optimization:

- Use ABD and UDOT to implement Neon sad_4d functions
- `vpx_variance32x32_neon`

We optimized the variance functions using the approach described in Calculating an average.

The following patch implements this optimization:

- Implement Neon variance functions using UDOT instruction

6.1 Results

The encode performance improved more than 17% at 1080p on the Neoverse N1 SDP platform. To achieve this, each optimization technique is combined and techniques are contributed back to the libvpx project.

Note: Results across Arm-based platforms depend on properties of the system, the compiler used, input and output resolution, and file and encode settings.

The following graph shows the performance results:

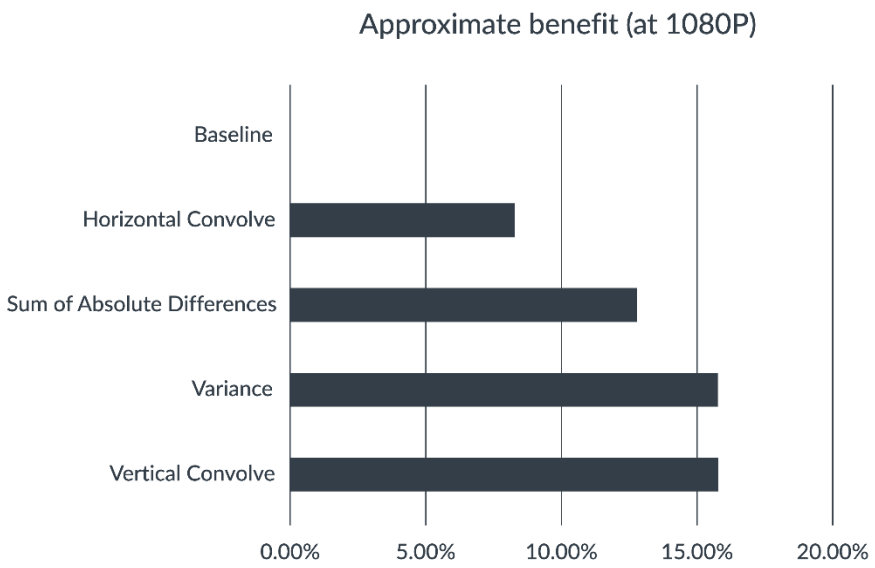


Figure 5: Performance results

7 Next steps

In this guide, we introduced three optimization techniques that use the dot product instructions from Armv8.4-A and shown how to use these techniques in a video encode library. The instructions improve performance by more than 15% on the latest processors. These techniques can apply across a range of workloads and increase the available throughput for widening multiply and accumulate for 8-bit data.

The next step is to learn more about how to optimize and use Arm Neon technology. See the Neon site for more information. This site contains examples of how to use SIMD architecture to unlock the performance of your devices.

8 Related information

The following resources are related to material in this guide:

- Arm Cortex-A series processors
- Armv8.4-A
- Neoverse
- Neoverse N1 SDP