



# Optimizing media pipelines using Armv8.x and Armv9.x features

Version 1.0

**Non-Confidential**

Copyright © 2024 Arm Limited (or its affiliates).  
All rights reserved.

**Issue 01**

110065\_0100\_01\_en



# Optimizing media pipelines using Armv8.x and Armv9.x features

Copyright © 2024 Arm Limited (or its affiliates). All rights reserved.

## Release information

### Document history

Issue	Date	Confidentiality	Change
0100-01	29 October 2024	Non-Confidential	1.0 Release

## Proprietary Notice

This document is protected by copyright and other related rights and the use 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 Limited ("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 the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, any patents, copyrights, trade secrets, trademarks, or other rights.

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, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

## 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.

## Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

## Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email [terms@arm.com](mailto:terms@arm.com).

# Contents

<b>1. Software codec optimization.....</b>	<b>6</b>
1.1 Arm64 ISA extensions for codec SIMD data processing.....	6
1.2 Compiler recommendations.....	7
1.3 Runtime feature detection for CPUs.....	7
1.4 Assembly code issues.....	8
<b>2. Optimization case studies.....</b>	<b>9</b>
2.1 libdav1d.....	9
2.2 libaom.....	10
2.3 x265.....	12
2.4 libyuv.....	14
<b>3. Optimization details.....</b>	<b>17</b>
3.1 Sum of Absolute Difference (SAD).....	17
3.2 Dot product.....	18
3.2.1 Sum of Absolute Difference (SAD).....	18
3.2.2 Sum of Squared Difference (SSD).....	19
3.3 Standard bitdepth convolution.....	19
3.3.1 Armv8.6 I8MM USDOT Optimization of SBD Convolution.....	20
3.3.2 Armv8.6 I8MM USMMLA Optimization of SBD Convolution.....	21
3.4 SVE2.....	21
3.4.1 SVE deployment in libaom, libvpx, and dav1d.....	22
3.5 Example optimization: x265 Sample Adaptive Offset (saoCuStats).....	23
3.5.1 x265 saoCuStats: Neon implementation.....	23
3.5.2 x265 saoCuStats: SVE implementation.....	24
3.5.3 x265 saoCuStats: SVE2 implementation.....	24
3.5.4 x265 saoCuStats: kernel performance results.....	25

# 1. Software codec optimization

This guide provides Arm guidance about how to use Armv8.x and Armv9.x features to optimize codecs and SIMD data processing algorithms.



Note

This is a fast-moving area of development, especially when considering specific benchmark results and optimizations. All information in this guide is correct as of October 2024.

Arm's approach to software codec optimization is as follows:

1. Optimize existing code:

- Think and design Arm-first to get the most out of the ISA. Many old Arm implementations are ported verbatim from x86, which does not necessarily produce the best results.
- Pay attention to, and optimize for, micro-architectural details as described in the [Software Optimization Guides on Arm Developer](#), for example the [Arm Cortex-A710 Core Software Optimization Guide](#).

2. Use additional Armv8.x and Armv9.x architectural features where beneficial:

- The Armv8.4 dot product and Armv8.6 l8MM 8-bit dot product and matrix multiply instructions result in a greater than 10% overall performance increase in multiple codecs.
- The Armv9 SVE and SVE2 16-bit dot product and histogram instructions give a significant performance increase over Neon.

Code can select Armv8.x and Armv9.x feature paths using runtime detection of CPU features with HWCAP. This means that the default build “just works” for different platforms.

## 1.1 Arm64 ISA extensions for codec SIMD data processing

The following table shows the available Arm64 ISA extensions for SIMD data processing that are most useful for codecs.



Note

Many other architectural features exist, but those shown in this table are the most beneficial for codecs.

Arm64 architectural feature	SIMD functionality	Linux HWCAP (recommended for runtime detection)	Compiler flag	Notes	First Cortex Arm CPU to support
FEAT_DotProd (v8.4)	Adds <code>UDOT</code> and <code>SDOT</code> 8-bit dot product	HWCAP_ASIMDDDP	<code>-march=armv8.2-a+dotprod</code>	Mandatory from Armv8.4	A55r1 / A75
FEAT_I8MM (v8.6)	Adds <code>USDOT</code> Adds 8x2*2x8 8-bit matrix multiply: <code>SMMLA</code> , <code>UMMLA</code> and <code>USMMLA</code>	HWCAP2_I8MM	<code>-march=armv8.2-a+i8mm</code>	Mandatory from Armv8.6	A510 / A710 / X1 (Arm TC2020)
FEAT_SVE	New variable vector length SIMD instruction set with predication. Adds complex numbers, 16-bit dot product, table lookup, gather load, and more.	HWCAP_SVE	<code>-march=armv8.2-a+sve</code>	Some server systems have SVE and Armv8.x (SVE with Armv8 only in server market)	A510 / A710 / X1 (Arm TC2020)
FEAT_SVE2 (v9)	Adds widening and narrowing DSP functionality as well as new histogram, match, and other instructions.	HWCAP2_SVE2	<code>-march=armv9-a+i8mm+sve2</code>	SVE2 is a superset of SVE, and only available from Armv9.0.	A510 / A710 / X2 (Arm TC2020)

## 1.2 Compiler recommendations

Arm recommends using the latest Clang or the latest GCC compiler for codec compilation.

Note in particular that GCC versions 10 and older give much worse performance than newer versions of both Clang and GCC. Arm does not recommended using these older GCC compilers.

SVE and SVE2 feature paths in `libaom`, `libvpx`, and `x265` require the intrinsics declared in `arm_neon_sve_bridge.h`. This header file is available in the following releases:

- Clang 17 and later.
- GCC 14 and later.

## 1.3 Runtime feature detection for CPUs

Runtime detection of CPU features means that code can select Armv8.x and Armv9.x feature paths based on available functionality without needing different builds for different platforms. Binaries for client devices must run on all existing and future hardware.

Arm recommends using the Linux HWCAP mechanism to check CPU features with the `getauxval()` function. Arm does not recommend checking `/proc/cpuinfo` because this method is less reliable. For more information, see the Arm Community blog [Runtime detection of CPU features on an ARMv8-A CPU](#).

Another technique for runtime feature detection is using Clang IFUNC to avoid function pointers. IFUNC can directly link to different versions of functions, depending on CPU features. Arm does

not currently recommend this technique on Android, because it introduces compiler version dependencies and is less friendly for upstreaming. However, using Clang IFUNC may be acceptable for software vendors who are able to use the latest compiler versions. This technique is supported in NDK r26+.

## 1.4 Assembly code issues

Developers writing assembly code should be aware of the following issues:

- Armv9 requires BTI landing pads for control flow integrity.

Android has enabled BTI for userspace, with v9 CPU. Any indirect branch to a register value must have a landing pad that matches the type of branch taken:

- `BTI c` for indirect function calls, for example `dav1d AARCH64_CALL_TARGET`.
- `BTI j` for non-function call branches, such as case statements, or “goto” jumps, for example `dav1d AARCH64_JUMP_TARGET`.

dav1d, included within Chrome or Chromium, has BTI enabled.

For more information, see the Arm Community blog [Enhancing Chromium’s Control Flow Integrity with Armv9](#).

- Execute-only code.

Android is moving towards support for execute-only code. This means constants must not be stored in `.text` regions.

For an example, see the dav1d commit [AArch64: Move constants of DotProd subpel filters to .rodata \(2355eeb8\)](#) and proposal [aarch64: Split the jump tables to a separate const section](#).

OpenBSD already supports execute-only code and requires that constants are not be stored in `.text` regions.



## 2. Optimization case studies

Recent Arm code optimization work includes the following:

- [libdav1d](#) (DotProd, I8MM, SVE2)
- [libaom](#) (CRC32, DotProd, I8MM, SVE2)
- [x265](#) (Neon, DotProd, I8MM, SVE, SVE2)
- [libyuv](#) (DotProd, I8MM, SVE2)

### 2.1 libdav1d

dav1d is an open-source AV1 cross-platform decoder developed by the VideoLAN and FFmpeg communities, and focused on speed and correctness. Arm has been contributing optimizations for both standard and high bitdepth algorithms since dav1d release 1.4.0.

Standard bitdepth optimizations include the following:

- Armv8.0, Armv8.4 DotProd, and Armv8.6 I8MM optimizations for sub-pixel filtering algorithms. Merged in release 1.4.2.
- Additional Armv8.6 I8MM optimizations for 6-tap sub-pixel filtering using `usmmla` matrix multiply instructions. `usmmla` matrix multiply instructions perform twice the work of corresponding `usdot` instructions. Merged in release 1.5.0.

The results of these standard bitdepth optimizations are as follows:

Core	1.4.2 vs 1.4.0 (fps)	1.5.0 vs 1.4.0 (fps)
Cortex-X3	+15%	+18%
Cortex-A715	+27%	+30%
Cortex-A510	+6%	+7%

High bitdepth optimizations include the following:

- Armv8.0 optimizations for sub-pixel filtering algorithms. Merged in release 1.4.2.
- Armv9.0 SVE2 optimizations for sub-pixel filtering using 16-bit dot-product instructions. This results in a 2x widening MAC throughput on big and mid cores. Merged in release 1.5.0.

The results of these high bitdepth optimizations are as follows:

Core	1.4.2 vs 1.4.0 (fps)	1.5.0 vs 1.4.0 (fps)
Cortex-X3	+7%	+13%
Cortex-A715	+12%	+18%
Cortex-A510	+3%	+3%

For both standard and high bitdepth results, performance was measured on a Google Pixel 8 Pro using the following corpus of 1080p videos taken directly from YouTube:

- [Models 1080p](#)
- [Balloons 1080p](#)
- [Mountain Bike 1080p](#)
- [Nature 1080p](#)
- [Vision Pro 1080p](#)
- [Georgia HDR](#)
- [RED HDR](#)

To replicate the dav1d full-decode performance results, do the following:

1. Convert input files to ivf format for use by the dav1d command-line decoder:

```
ffmpeg -i <av1_input.mp4> -c:v copy <av1_output.ivf>
```

2. Run the benchmark:

```
tools/dav1d --threads 1 --muxer null -i <av1_input.ivf>
```

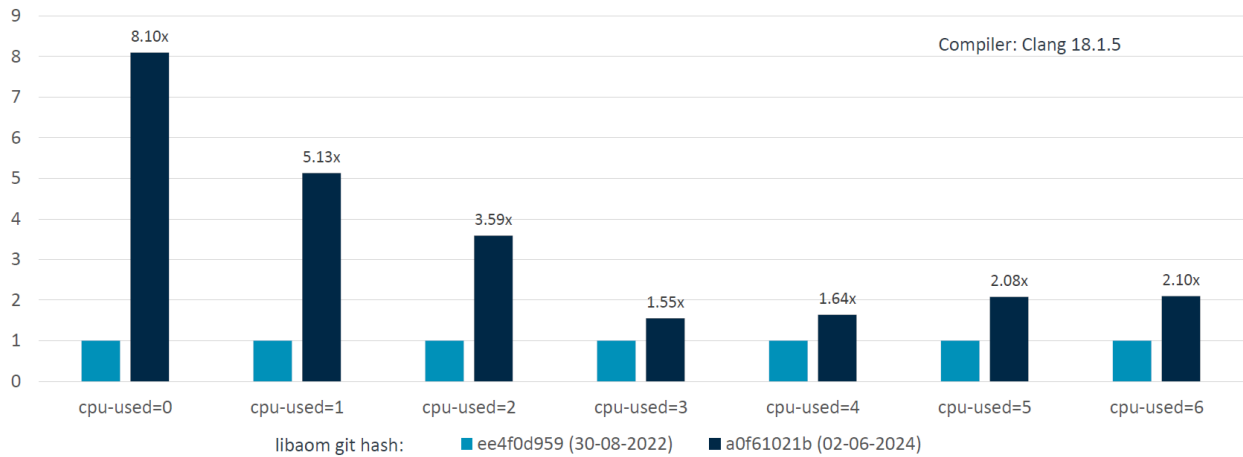
For more information, see the commit messages on merge request [1632- AArch64: Add DotProd support for convolutions](#).

## 2.2 libaom

Libaom is an open-source video codec library developed by AOMedia. It serves as the reference implementation of the AV1 video compression format.

The following figure shows the performance increase of libaom standard bitdepth 1080p VoD encoding on Graviton 4 (single thread) with Arm optimizations:

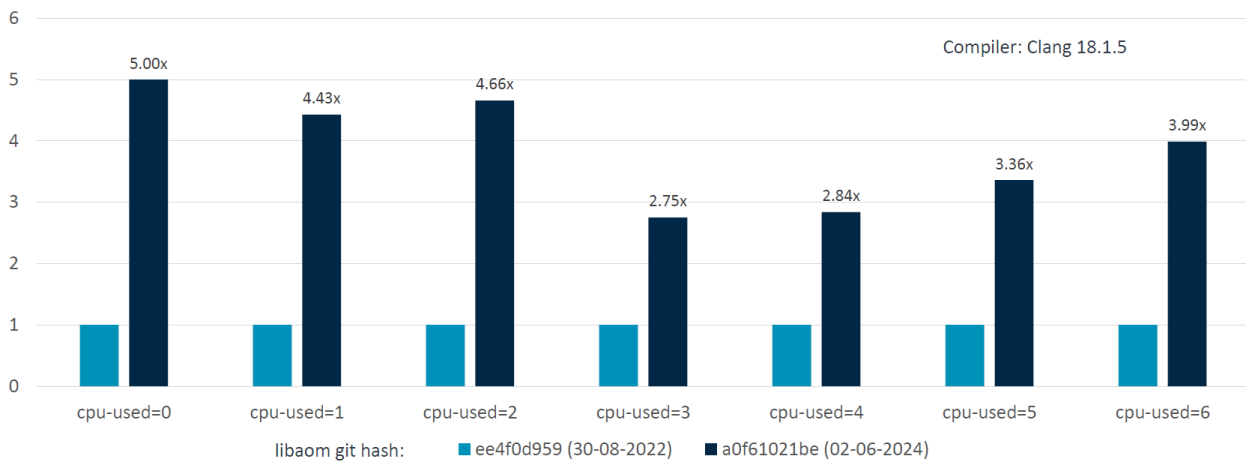
**Figure 2-1: libaom standard bitdepth 1080p VoD with Arm optimizations**



cpu-used selects complexity of encode, where 0 means slow and 10 means fast.

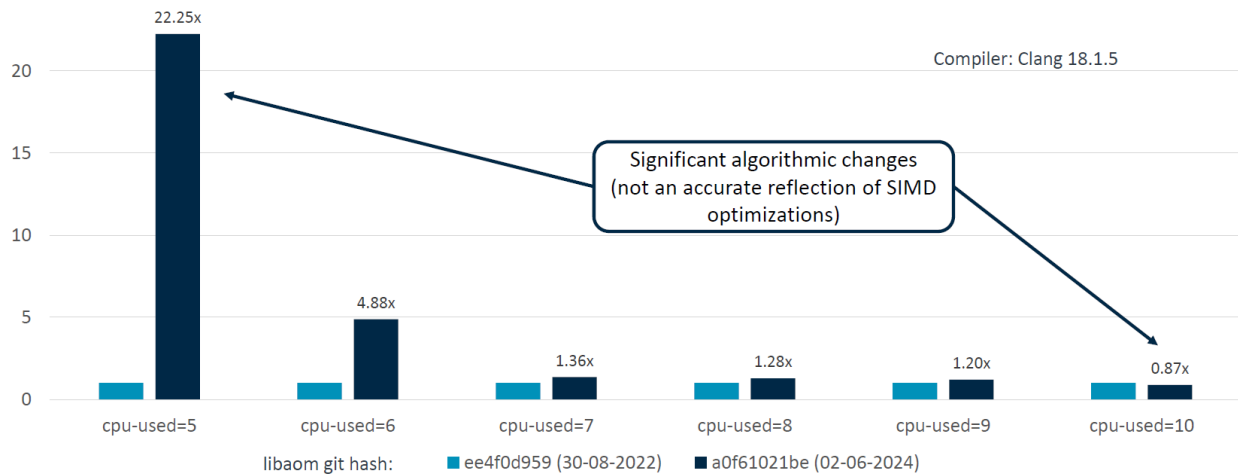
The following figure shows the performance increase of libaom high bitdepth 4K VoD encoding on Graviton 4 (single thread) with Arm optimizations:

**Figure 2-2: libaom high bitdepth 4K VoD with Arm optimizations**



The following figure shows the performance increase of libaom standard bitdepth 1080p live-stream encoding on Graviton 4 (single thread) with Arm optimizations:

**Figure 2-3: libaom standard bitdepth 1080p live-stream with Arm optimizations**



To replicate the libaom performance results, use the following commands:

- VoD:
  - `./aomenc --good --cpu-used=${CPU_USED} --bit-depth=8 -o output.mkv ${VIDEO_SBD_FHD}`
  - `./aomenc --good --cpu-used=${CPU_USED} --bit-depth=10 -o output.mkv ${VIDEO_HBD_4K}`
- Live-stream:
  - `./aomenc --rt --cpu-used=${CPU_USED} --bit-depth=8 -o output.mkv ${VIDEO_SBD_FHD}`

See the performance results figures above for the CPU\_USED range and the libaom hashes.

Video inputs are available as follows:

- 8-bit 1080p: [https://ultravideo.fi/video/Bosphorus\\_1920x1080\\_120fps\\_420\\_8bit\\_YUV\\_Y4M.7z](https://ultravideo.fi/video/Bosphorus_1920x1080_120fps_420_8bit_YUV_Y4M.7z)
- 10-bit 4K: [https://ultravideo.fi/video/Bosphorus\\_3840x2160\\_120fps\\_420\\_10bit\\_YUV\\_Y4M.7z](https://ultravideo.fi/video/Bosphorus_3840x2160_120fps_420_10bit_YUV_Y4M.7z)

## 2.3 x265

x265 is a free software library for encoding digital video streams in the H.265/MPEG-H HEVC compression format.

Arm has developed and merged upstream the following x265 optimization patches as of x265 release 4.0:

- Initial refactoring and bug fixes
- Vectorization of saoCuStats (Neon, SVE and SVE2)

- SAD/SADxN optimizations
- SSE optimizations
- Quantization optimizations
- Enable `-flax-vector-conversions=none` for Neon intrinsics code
- Enable `-werror` for Neon intrinsics code
- DCT optimizations
- Convolution optimizations

Performance results for x265 4.0 versus x265 3.6 are as follows:

Preset	x265 v3.6	After all Arm patches	FPS uplift (%)
Ultrafast	29.34	38.16	30.06
Superfast	23.12	29.89	29.28
Veryfast	11.66	18.32	57.12
Faster	11.60	18.11	56.12
Fast	6.55	10.00	52.67
Medium	8.43	12.22	44.96
Slow	2.37	3.34	40.93
Slower	0.54	0.72	33.33
Veryslow	0.31	0.42	35.48
Average			42.22

These performance results were obtained as follows:

- Platform: AWS Graviton 4 (Neoverse V2)
- OS: Ubuntu 24.04
- Compiler: Clang 18.1.5
- Input video: [Bosphorus 1080p](#)
- Numbers quoted are single-thread performance on a lightly-loaded system
- All encodings are standard bitdepth (8-bit)

To replicate the x265 performance results, use the following command:

```
./x265 --preset ${PRESET} --input ${VIDEO_SBD_FHD} --output output.mkv --pools 1 --
frame-threads 1 --no-wpp
```

Where: - `${PRESET}` is a value from the table above. - The workload is [https://ultravideo.fi/video/Bosphorus\\_1920x1080\\_120fps\\_420\\_8bit\\_YUV\\_Y4M.7z](https://ultravideo.fi/video/Bosphorus_1920x1080_120fps_420_8bit_YUV_Y4M.7z).

Note that x265 currently uses compile-time CPU selection, although there are plans to update to run-time selection soon.

Arm optimizations started after x265 release 3.6.

## 2.4 libyuv

libyuv is an open source project that provides YUV scaling and conversion functionality, including the following:

- Scale YUV to prepare content for compression, with point, bilinear or box filter.
- Convert to YUV from webcam formats for compression.
- Convert to RGB formats for rendering/effects.
- Rotate by 90/180/270 degrees to adjust for mobile devices in portrait mode.

libyuv is used in many different applications, including the following:

- Chromium WebRTC
- Google Meet
- ChromeOS Camera
- Android Media

You can use Android Simpleperf to establish when libyuv is being used, by obtaining a profile, searching for the `libyuv_` prefix, and identifying the name of the specific kernel being used.

Arm started optimization work on libyuv in February 2024, making significant performance uplifts to libyuv by optimizing Neon code and pushing further with SVE2.

For example, Arm made optimizations to the following frame buffer post-processing functions for 10-bit dav1d:

- `convertPlanar16ToY410OrRGBA1010102()`
  - `libyuv::I420ToAB30Matrix()`
  - `libyuv::I210ToAB30Matrix()`

Previously, these APIs did not use Arm SIMD instructions.

The following table shows the optimization results for `I420ToAB30Matrix`:

Core	C -> Neon	Neon -> SVE2
Cortex-X4	8.15x	1.08x
Cortex-A720	6.93x	1.27x
Cortex-A520	4.74x	1.06x

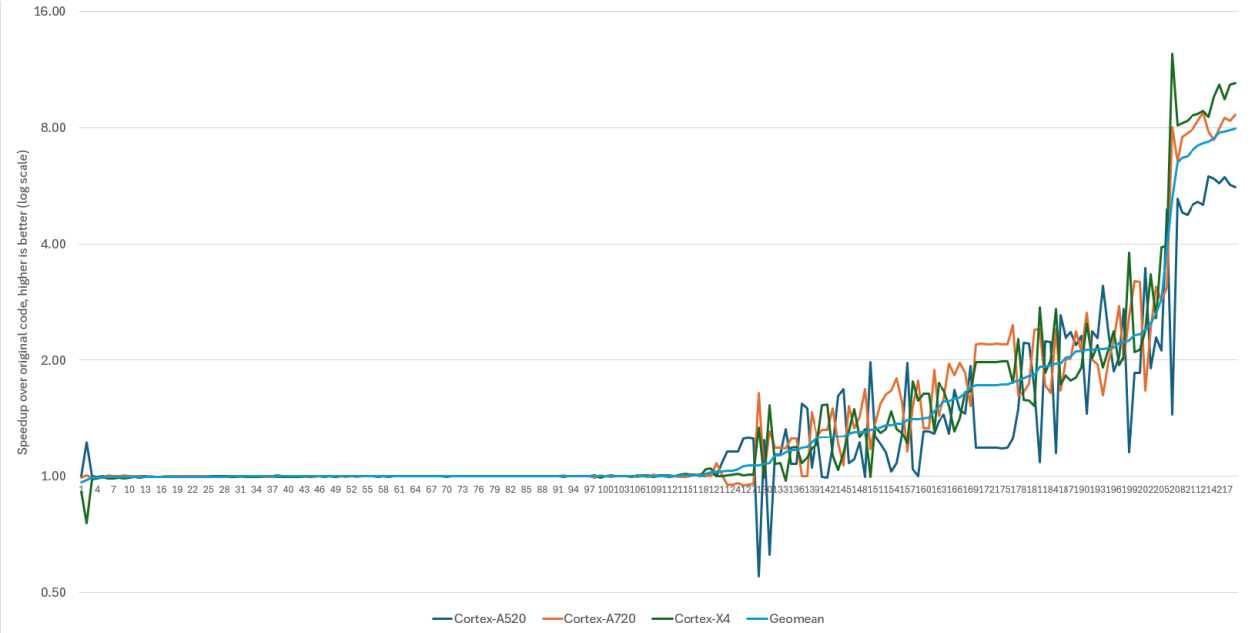
The following table shows the optimization results for `I210ToAB30Matrix`:

Core	C -> Neon	Neon -> SVE2
Cortex-X4	7.76x	1.06x
Cortex-A720	6.36x	1.19x

Core	C -> Neon	Neon -> SVE2
Cortex-A520	4.63x	1.04x

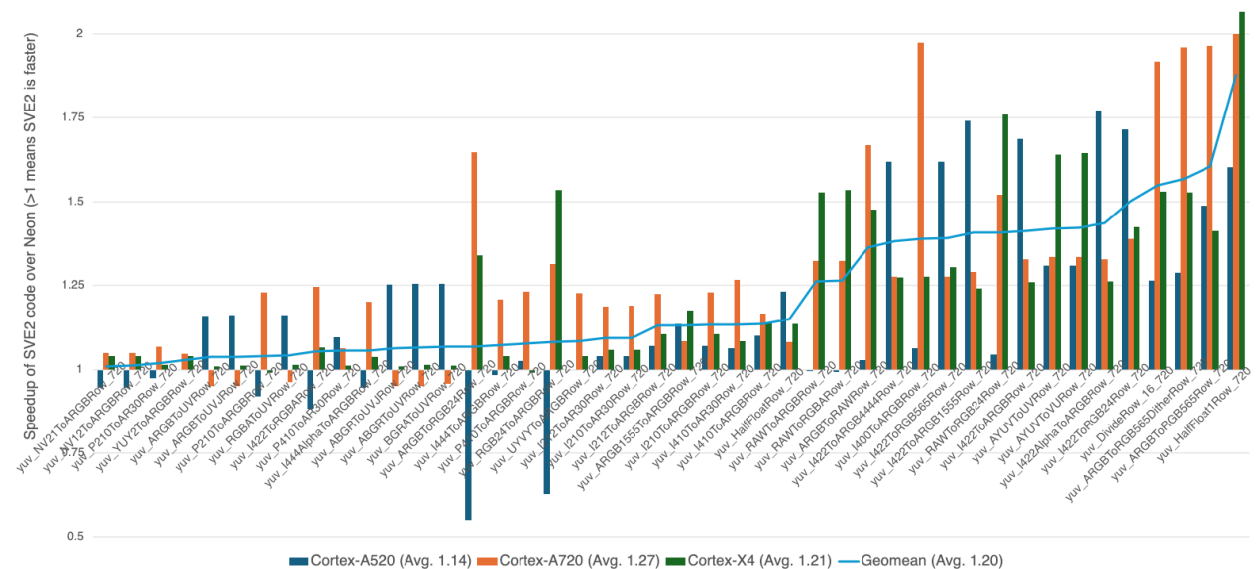
The following chart shows the performance uplift for all 217 libyuv performance tests along the X axis, including SVE2 optimizations (only a subset of kernels currently), sorted by speedup for clarity. Optimization work is continuing as further libyuv kernels are optimized. Large gains are for previously unoptimized kernels where there was no existing Neon, for example `I420ToAB30Matrix()`.

**Figure 2-4: Libyuv latest kernel speedups compared to February 2024, sorted by geomean speedup (higher is better)**



The following chart shows more detail of the right-hand side of the previous chart, showing specific uplift of SVE2 optimizations from a Neon/DotProd/I8MM optimized baseline.

**Figure 2-5: Libyuv latest SVE2 kernel speedups over latest Neon, sorted by geomean speedup**



To replicate the libaom performance results, do the following:

- Compare upstream against the start of Arm's optimization work, which includes Neon improvements, DotProd, I8MM, and SVE2.
  - Arm benchmarking was done starting from commit `b66c42d4a8fbf56d2c83aa3ea55761b9fef363f5` Thu Feb 29 2024.
  - Optimization work is ongoing.
- Runtime capability checking uses `getauxval()` and HWCAP.

For more information about features and feature detection, see [libyuv documentation: Feature detection on AArch64](#).

- To run individual tests, use the following commands:

```
LIBYUV_CPU_INFO=7
LIBYUV_REPEAT=1000
./libyuv_unittest --gtest_filter="LibYUVConvertTest.I420ToARGB_Any" | grep OPT
3211 us C - 372 us OPT
LIBYUV_CPU_INFO=247
LIBYUV_REPEAT=1000
./libyuv_unittest --gtest_filter="LibYUVConvertTest.I420ToARGB_Any" | grep OPT
3206 us C - 296 us OPT
```

372 us / 296 us shows that Neon+DotProd+I8MM+SVE2 is 25.7% faster than Neon.

These commands use the `libyuv_unittest` CPU feature selection. The meanings of the `LIBYUV_CPU_INFO` bitmasks are as follows:

- 7: Neon only
- 247: Neon+DotProd+I8MM+SVE2



## 3. Optimization details

This section of the guide provides detail on the specific instructions used in the various different codec optimizations described in [Optimization case studies](#).

The optimizations described are as follows:

- Sum of Absolute Difference (SAD)
- Dot product
- Standard bitdepth convolution
- SVE2

### 3.1 Sum of Absolute Difference (SAD)

Libvpx, libaom, x264, and x265 SAD kernels used long chains of `UABAL` and `UABAL2` instructions.

The original x265 SAD code was as follows:

```
.macro SAD_32
    ldl    {v0.16b-v1.16b}, [x0], x1
    ldl    {v2.16b-v3.16b}, [x2], x3
    ldl    {v4.16b-v5.16b}, [x0], x1
    ldl    {v6.16b-v7.16b}, [x2], x3
    uabal1 v16.8h, v0.8b, v2.8b
    uabal2 v17.8h, v0.16b, v2.16b
    uabal1 v18.8h, v1.8b, v3.8b
    uabal2 v19.8h, v1.16b, v3.16b
    uabal1 v16.8h, v4.8b, v6.8b
    uabal2 v17.8h, v4.16b, v6.16b
    uabal1 v18.8h, v5.8b, v7.8b
    uabal2 v19.8h, v5.16b, v7.16b
```

The following table shows the theoretical performance of this code on a Cortex-X1:

Instruction Group	AArch64 Instructions	Execution Latency	Execution Throughput	Utilized Pipelines
ASIMD absolute diff accum long	SABAL(2), UABAL(2)	4(1)	2	V13

This shows that the `UABAL` and `UABAL2` instructions can only execute on half of the Neon pipes in Arm-designed out-of-order CPUs. Because SAD is one of the highest utilized algorithms in most codecs, wasting half the Neon execute bandwidth has a significant impact on efficiency. For more information about the contents of this table, see the [Arm Cortex-X1 Core Software Optimization Guide](#).

The optimized code, performing SAD using the full Neon bandwidth, is as follows:

```
.macro SAD_32
    ldl    {v0.16b-v1.16b}, [x0], x1
    ldl    {v2.16b-v3.16b}, [x2], x3
    ldl    {v4.16b-v5.16b}, [x0], x1
```

```
ld1    {v6.16b-v7.16b}, [x2], x3
uabd   v0.16b, v0.16b, v2.16b
uabd   v1.16b, v1.16b, v3.16b
uabd   v4.16b, v4.16b, v6.16b
uabd   v5.16b, v5.16b, v7.16b
uadalp v16.8h, v0.16b
uadalp v17.8h, v1.16b
uadalp v18.8h, v4.16b
uadalp v19.8h, v5.16b
```

The following table shows the theoretical performance of this code on a Cortex-X1:

Instruction Group	AArch64 Instructions	Execution Latency	Execution Throughput	Utilized Pipelines
ASIMD absolute diff	SABD, UABD	2	4	V
ASIMD pairwise add and accumulate long	SADALP, UADALP	4(1)	2	V

In this optimized code, half of the Neon pipes execute `UABD` instructions while the other half execute `UADALP` instructions. This results in 2x throughput compared to the original implementation. For more information about the contents of this table, see the [Arm Cortex-X1 Core Software Optimization Guide](#).

Identifying this optimization opportunity relies on familiarity with the Software Optimization Guide (SWOG).

## 3.2 Dot product

Armv8.4 introduced the dot product instruction `UDOT`. This instruction has applications beyond just the calculation of dot products, and is often a useful optimization tool. This section describes two situations where `UDOT` can optimize algorithms that do not seem initially to be related to dot products.

### 3.2.1 Sum of Absolute Difference (SAD)

`UDOT` is very useful as a widening-add instruction:

- To use, simply `UDOT` a vector with `#1`.
- 8-bit to 32-bit widening is very useful.
- Gives the same throughput as other basic Neon arithmetic.

SAD original instructions:

```
UABAL
UABAL2
```

SAD optimized instructions:

```
UABD
```

**UDOT**

Using `UABD` and `UDOT` instructions results in 2x the throughput for SAD with the bonus of a wider 32-bit accumulator. This is because `UABAL` can only execute on half the Neon pipes in Arm cores, while `UABD` and `UDOT` can execute on all of them.

### 3.2.2 Sum of Squared Difference (SSD)

To calculate SSD, `UDOT` a vector of data with itself. 8-bit to 32-bit widening is again very useful.

SSD original instructions:

```
SABDL
SABDL2
SMLAL
SMLAL2
SMLAL
SMLAL2
```

SSD optimized instructions:

```
UABD
UDOT
```

This optimization can be very useful because these calculations are the basis for variance, subpixel-variance, SSE and temporal filter algorithms.

## 3.3 Standard bitdepth convolution

For 8-bit standard bitdepth convolution, the throughput of the Neon multiply-and-accumulate instructions is the performance bottleneck.

For example, consider the following example code from the libaom/libvpx convolution kernel:

```
static INLINE uint8x8_t convolve8(const int16x8_t s0, const int16x8_t s1,
                                   const int16x8_t s2, const int16x8_t s3,
                                   const int16x8_t s4, const int16x8_t s5,
                                   const int16x8_t s6, const int16x8_t s7,
                                   const int16x8_t filter) {
    int16x8_t sum = vmulq_laneq_s16(s0, filter, 0);
    sum = vmlaq_laneq_s16(sum, s1, filter, 1);
    sum = vmlaq_laneq_s16(sum, s2, filter, 2);
    sum = vmlaq_laneq_s16(sum, s5, filter, 5);
    sum = vmlaq_laneq_s16(sum, s6, filter, 6);
    sum = vmlaq_laneq_s16(sum, s7, filter, 7);
    sum = vqaddq_s16(sum, vmulq_laneq_s16(s3, filter, 3);
    sum = vqaddq_s16(sum, vmulq_laneq_s16(s4, filter, 4);

    return vqshrshrun_n_s16(sum, FILTER_BITS);
}
```

All source data in this example is `uint8_t`, but we need to widen to `int16_t` for signedness.

The following table shows the theoretical performance of this code on a Cortex-X1:

Instruction Group	AArch64 Instructions	Execution Latency	Execution Throughput
ASIMD multiply accumulate	MLA, MLS	4(1)	2

Cortex-X1 can only execute 2 Neon MLA instructions per cycle. Given the dependency chain, this means that the multiplication part of the example convolution kernel takes 8 cycles, excluding result end latency. Executing 2 of these kernels in parallel gives a maximum theoretical throughput of 16 source element multiply-accumulates per cycle.

### 3.3.1 Armv8.6 I8MM USDOT Optimization of SBD Convolution

The following code shows the same kernel optimized to use the Armv8.6 `USDOT` instruction:

```
static INLINE uint8x8_t convolve8(const uint8x16_t samples
                                const int8x8_t filter,
                                const uint8x16x3_t permute_tbl) {
    // Permute samples ready for dot product
    // { 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6 }
    // { 4, 5, 6, 7, 5, 6, 7, 8, 6, 7, 8, 9, 7, 8, 9, 10 }
    // { 8, 9, 10, 11, 9, 10, 11, 12, 10, 11, 12, 13, 11, 12, 13, 14 }
    uint8x16_t permuted_samples[3] = { vqtbl1q_u8(samples, permute_tbl.val[0]),
                                       vqtbl1q_u8(samples, permute_tbl.val[1]),
                                       vqtbl1q_u8(samples, permute_tbl.val[2]) };

    int32x4_t sum0123 =
        vusdotq_lane_s32(vdupq_n_s32(0), permuted_samples[0], filter, 0);
    sum0123 = vusdotq_lane_s32(vdupl_n_s32(0), permuted_samples[1], filter, 1);

    int32x4_t sum4567 =
        vusdotq_lane_s32(vdupq_n_s32(0), permuted_samples[1], filter, 0);
    sum4567 = vusdotq_lane_s32(vdupl_n_s32(0), permuted_samples[2], filter, 1);

    // Narrow and re-pack
    int16x8_t sum = vcombine_s16(vshrn_n_s32(sum0123, 1), vshrn_n_s32(sum4567, 1));
    return vqshrln_n_16(sum, FILTER_BITS - 1);
}
```

The following table shows the theoretical performance of this code on a Cortex-X1:

Instruction Group	AArch64 Instructions	Execution Latency	Execution Throughput
ASIMD dot product using signed and unsigned integers	SUDOT, USDOT	3(1)	4

Cortex-X1 can execute 4 Neon `USDOT` instructions per cycle. This gives a maximum theoretical throughput of 32 source element multiply-accumulate per cycle – twice that of the original code.

This optimized example also operates on 8-bit elements, but with a small extra cost for in-kernel data re-arrangement.

### 3.3.2 Armv8.6 I8MM USMMLA Optimization of SBD Convolution

The `USMMLA` instruction does twice the work of `USDOT`, but requires space to stagger the filter.

The code in [Armv8.6 I8MM USDOT Optimization of SBD Convolution](#) showed the use of `USDOT` with 8-tap filters.

The following code shows the same kernel optimized to use the Armv8.6 I8MM `USMMLA` instruction with 6-tap filters:

```
// Staggered filter for use with the matrix multiply instructions:
// { f0, f1, f2, f3, f4, f5, 0, 0, 0, f0, f1, f2, f3, f4, f5, 0 }
static INLINE uint8x8_t convolve6(const uint8x16_t samples
                                const int8x16_t filter,
                                const uint8x16x2_t permute_tbl) {
    // Permute samples ready for dot product
    // { 0, 1, 2, 3, 4, 5, 6, 7, 2, 3, 4, 5, 6, 7, 8, 9 }
    // { 4, 5, 6, 7, 8, 9, 10, 11, 6, 7, 8, 9, 10, 11, 12, 13 }
    uint8x16_t permuted_samples[2] = { vqtbl1q_u8(samples, permute_tbl.val[0]),
                                       vqtbl1q_u8(samples, permute_tbl.val[1]) };

    // These instructions multiply a 2x8 matrix (samples) by an 8x2 matrix
    // (filter), destructively accumulating into the destination register
    int32x4_t sum0123 = vusmmlaq_s32(vdupq_n_s32(0), permuted_samples[0], filter, 0);
    int32x4_t sum4567 = vusmmlaq_s32(vdupq_n_s32(0), permuted_samples[1], filter, 0);

    // Narrow and re-pack
    int16x8_t sum = vcombine_s16(vshrn_n_s32(sum0123, 1), vshrn_n_s32(sum4567, 1));
    return vqrshrun_n_16(sum, FILTER_BITS - 1);
}
```



Many 8-tap filters are zero padded 5-, 6-, or 7-tap filters.

## 3.4 SVE2

SVE2, introduced in Armv9, can provide useful performance gains over Neon in some codec algorithms.

As a general principle, unless you can leverage some functionality in SVE that fundamentally does not exist in Neon, Neon and SVE code will have similar performance at the same vector length.

Examples of functionality that is unique to SVE and SVE2 include the following:

- 16-bit dot product instructions introduced in SVE, which were not backported to Neon.
- 16-bit SVE dot-product instructions provide a 2x widening multiply-accumulate bandwidth increase over Neon `SMLAL` and `UMLAL` at 128-bit VL.

### 3.4.1 SVE deployment in libaom, libvpx, and dav1d

Dot product instructions operating on 16-bit input elements are exclusive to the SVE instruction set. However, we can access these instructions from a predominantly Neon context by making use of the Neon-SVE bridge intrinsics to re-interpret Neon vectors as SVE vectors, with the high part of the SVE vector being “don’t care” if the SVE vector is longer than 128 bits.

While this technique is sub-optimal on machines that have an SVE vector length greater than 128 bits, because the remainder of the vector is unused, this approach is still beneficial when compared to a Neon-only solution.

The following code shows this technique:

```
static INLINE uint64x2_t vpx_dotq_u16(uint64x2_t acc, uint16x8_t x,
                                     uint16x8_t y) {
    return svget_neonq_u64(svdot_u64(svset_neonq_u64(svundef_u64(), acc),
                                     svset_neonq_u16(svundef_u16(), x),
                                     svset_neonq_u16(svundef_u16(), y)));
}
```

This technique is possible because SVE and Neon vectors are banked, occupying the same silicon registers. This allows us to re-interpret Neon registers as SVE ones and vice-versa to use useful SVE instructions.

Performance uplift results for high bitdepth decode optimization with SVE2 are as follows: - +10% dav1d (AV1) - +8% libvpx (VP9)

The following results show the SVE uplift over Neon on Neoverse V2, analyzing libvpx variance which is one example of many algorithms that benefit:

**Figure 3-1: Analyzing libvpx variance using Neon vs SVE**

	neon ->	sve	
vpx_highbd_10_variance4x4:	7.91 ->	7.76 (0.981x)	cycles per iteration
vpx_highbd_10_variance4x8:	13.10 ->	13.72 (1.047x)	cycles per iteration
vpx_highbd_10_variance8x4:	9.89 ->	7.53 (0.761x)	cycles per iteration
vpx_highbd_10_variance8x8:	17.04 ->	14.07 (0.826x)	cycles per iteration
vpx_highbd_10_variance8x16:	30.53 ->	26.44 (0.866x)	cycles per iteration
vpx_highbd_10_variance16x8:	29.21 ->	24.83 (0.850x)	cycles per iteration
vpx_highbd_10_variance16x16:	60.58 ->	46.49 (0.767x)	cycles per iteration
vpx_highbd_10_variance16x32:	120.95 ->	87.02 (0.719x)	cycles per iteration
vpx_highbd_10_variance32x16:	114.61 ->	81.68 (0.713x)	cycles per iteration
vpx_highbd_10_variance32x32:	231.27 ->	146.02 (0.631x)	cycles per iteration
vpx_highbd_10_variance32x64:	450.64 ->	274.40 (0.609x)	cycles per iteration
vpx_highbd_10_variance64x32:	507.24 ->	307.23 (0.606x)	cycles per iteration
vpx_highbd_10_variance64x64:	1013.45 ->	598.98 (0.591x)	cycles per iteration
Geomean: 0.755 (lower is better)			

## 3.5 Example optimization: x265 Sample Adaptive Offset (saoCuStats)

Sample adaptive offset (SAO) is a new feature introduced in H.265, to reduce artifacts and distortions like noise and ringing around edges, and to improve the visual quality. The aim of SAO is to reduce sample distortion by first classifying reconstructed samples into different categories, obtaining an offset for each category, and then adding the offset to each sample of the category. This results in improved subjective quality of reconstructed pictures.

One of the compute-intensive stages of the SAO algorithm is collecting the statistics for the categories, which has been converted to SIMD.

### 3.5.1 x265 saoCuStats: Neon implementation

The statistics collection phase of the SAO algorithm, saoCuStats, is suited to a vectorized implementation. The histogram computation involves a small number of classes, just 5, which enables an algorithm using compare instructions.

The following code shows the scalar algorithm pseudocode:

```
for i in 0..N
    input_class = compute_class(input[i])
    for j in 0..5
        count[j] += (input_class == j)
        diff[j] += (input_class == j) * diff[i]
```

The following code shows a Neon implementation of the algorithm:

```
/*
 * Compute Edge Offset statistics (count and stats)
 * To save some instructions compute count and stats as negative values - since
 * output of Neon comparison instructions for a true condition is all 1s (-1)
 */
static inline void compute_eo_stats(const int8x16_t edge_type,
                                   int16x8_t diff_lo, int16x8_t diff_hi,
                                   int16x8_t *count, int32x4_t *stats)
{
    int8x16_t mask[5];
    int16x8_t mask_lo[5];
    int16x8_t mask_hi[5];
    int16x8_t tmp_stats[5];

    for (int i = 0; i < 5; ++i) {
        // Create a mask for each edge type.
        mask[i] = vreinterpretq_s8_u8(vceqq_s8(edge_type, vdupq_n_s8(i)));

        // Compute negative counts for each edge type
        count[i] = vpaddlq_s8(count[i], mask[i]);

        // Widen the masks to 16-bit.
        mask_lo[i] = vreinterpretq_s16_s8(vzip1q_s8(mask[i], mask[i]));
        mask_hi[i] = vreinterpretq_s16_s8(vzip2q_s8(mask[i], mask[i]));

        // Compute negative stats for each edge type.
        tmp_stats[i] = vmulq_s16(diff_lo, mask_lo[i]);
        tmp_stats[i] = vmlaq_s16(tmp_stats[i], diff_hi, mask_hi[i]);
    }
}
```

```

    stats[i] = vpadalq_s16(stats[i], tmp_stats[i]);
}
}

```

The inner loop of the kernel processes 16 input values at once using Neon instructions. Each iteration generates a mask for each edge type, creating a histogram bucket. This mask is used to accumulate result into separate buffers for each edge type. For efficiency, the code computes count and stats as negative values, since the output of Neon comparison instructions for a true condition is all 1s (-1).

### 3.5.2 x265 saoCuStats: SVE implementation

Using the SVE 16-bit dot-product instructions gives a 2x widening MAC bandwidth uplift.

The following code shows how the Neon implementation of the `compute_eo_stats` function can be optimized to use the SVE 16-bit dot product:

```

// Neon-SVE bridge helper
static inline int64x2_t x265_sdotq_s16(int64x2_t acc, int16x8_t x, int16x8_t y)
{
    return svget_neonq_s64(svdot_264(svset_neonq_s64(svundef_s64(), acc),
                                           svset_neonq_s16(svundef_s16(), x),
                                           svset_neonq_s16(svundef_s16(), y)));
}
.
.
.
// Compute negative stats for each edge type using SDOT.
stats[i] = x265_sdotq_s16(stats[i], diff_lo[i], mask_lo[i]);
stats[i] = x265_sdotq_s16(stats[i], diff_hi[i], mask_hi[i]);
.
.
.

```

### 3.5.3 x265 saoCuStats: SVE2 implementation

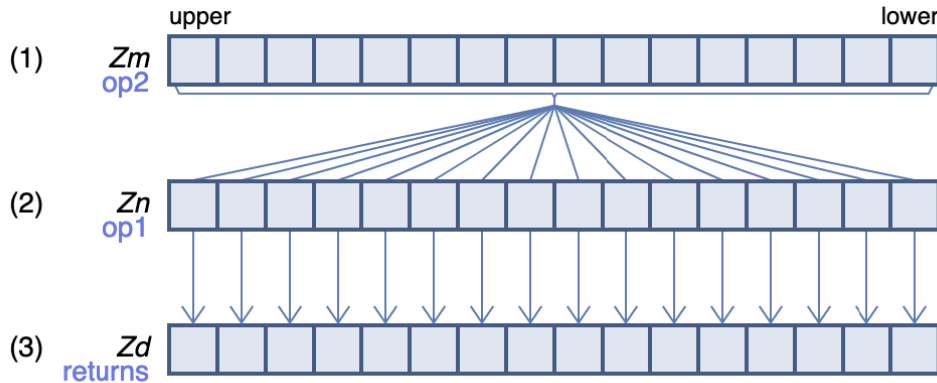
Using SVE2, the algorithm can be optimized using the `HISTSEG` instruction to count matching elements in vector segments.

The following diagram shows the operation of the `HISTSEG` instruction. For each 8-bit element in vector 2, count the number of equal elements from vector 1 within the same 128-bit segment, and write the count to vector 3.



**Figure 3-2: HISTSEG instruction**

## 128-bit SVE



The following code shows the implementation of count computation using the SVE2 HISTSEG instruction:

```
static inline uint8x16_t sve_count(int8x16_t in)
{
    // We do not care about initializing the values in the rest of the vector,
    // for VL > 128, as HISTSEG counts matching elements in 128-bit segments
    svint8_t edge_type = svset_neonq_s8(svundef_s8(), in);

    // Use an arbitrary value outside of range [-2, 2] for lanes we don't
    // need to use the result from.
    const int DC = -3;
    // s_eoTable maps edge types to memory in order: {2, 0, 1, 3, 4}
    // We use (edge_class - 2) resulting in: {0, -2, -1, 1, 2}
    int8x16_t idx = {0, -2, -1, 1, 2, DC, DC, DC, DC, DC, DC, DC, DC, DC, DC, DC, DC};
    svint8_t svidx = svset_neonq_s8(svundef_s8(), idx);

    svuint8_t count = svhistseg_s8(svidx, edgetype);
    return svget_neonq_u8(count);
}
```

### 3.5.4 x265 saoCuStats: kernel performance results

Originally, this kernel was a scalar C implementation on Arm platforms.

The following Arm commits add Neon implementations for standard and high bitdepth:

- [c07b3ae](#) - AArch64: Add Neon saoCuStats primitives for low bitdepth
- [f29dc45](#) - AArch64: Add Neon saoCuStats primitives for high bitdepth

The following Arm commit adds SVE, with 16-bit dot-product instructions giving 2x widening MAC throughput over Neon at same vector length:

- [ad1a30a](#) - AArch64: Add SVE saoCuStats primitives

The following Arm commit adds SVE2, with histogram instructions counting matching elements in a vector:

- [da8443b - AArch64: Add SVE2 saoCuStats primitives](#)

The following table shows the performance uplift on a Neoverse V2 machine building with Clang 19:

	C -> Neon	Neon -> SVE	SVE -> SVE2
saoCuStatsE0	2.94x	1.10x	1.11x
saoCuStatsE1	3.59x	1.11x	1.07x
saoCuStatsE2	3.42x	1.13x	1.12x
saoCuStatsE3	3.41x	1.12x	1.12x