Optimization of x265 encoder using ARM SVE

AOKI Ryosuke, MURAO Hirokazu (Advisor)

Abstract

We optimized several heavily used functions in x265, an open source implementation of H.265/HEVC, using SVE instructions. The result showed that our implementation reduced the number of instructions executed up to 50% compared to the code generated by GCC using the original C++ source code. The implementation also scales better with the vector length than the original code.

Background

Scalable Vector Extension (SVE)

Scalable Vector Extension (SVE) [4] is a vector extension for ARMv8-A, a 64bit CPU architecture developed by ARM Ltd. The most notable feature of SVE is its scalable vector registers. SVE does not specify the exact length of these registers, but let the implementation choose the length, from 128 bits up to 2048 bits. SVE adopts the vector-length agnostic (VLA) programming model, making it possible to run programs on every SVE platform with different vector length, without the need of recompiling. In this April, ARM announced SVE2, a new extension based on SVE, which has new instruction to vectorize DSP and multimedia SIMD codes [3].

H.265/HEVC

H.265/HEVC is a video codec developed and standardized by Joint Collaborative Team on Video Coding (JCT-VC) on 2013. H.265/HEVC achieved higher compression efficiency compared to its predecessor H.264, at the expense of significantly higher computational cost.

x265

x265 [2] is a open source implementation of H.265/HEVC encoder, developed by MulticoreWare. It is written in C++ and assembly. Several frequently used subroutines, called “primitives” are implemented in assembly but it is not available for 64bit ARM architecture (AArch64).

Implementation

Determining cost of “primitives”

To find out most heavily used part in the x265 program, we profiled an execution of a test video encoding, using an AArch64 machine (without SVE instructions). The pie chart on the right shows the result of profiling. The overheads (measured in number of instructions executed) from top 10 primitives sum up to about 2/3 of total number of executed instructions. All of these functions are called “primitives” in x265 source code; primitives process block(s) and return a block or a integer value. “Blocks” are rectangle parts of the picture, and their size varies from 4x4 to 64x64. Blocks contain 8 bit or 16 bit integers, not floats.

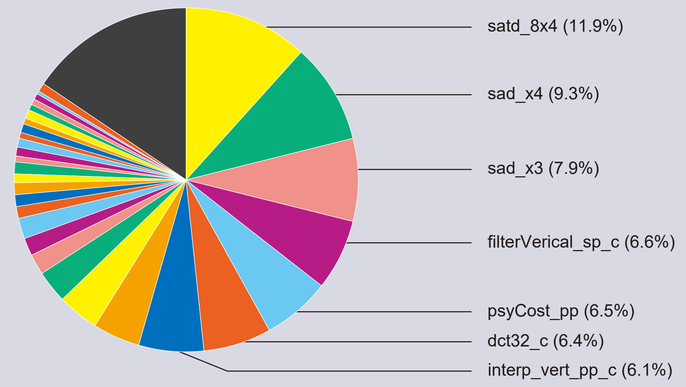

Figure 1. Number of instructions executed measured in encoding a clip using x265

Optimization example 1: SATD

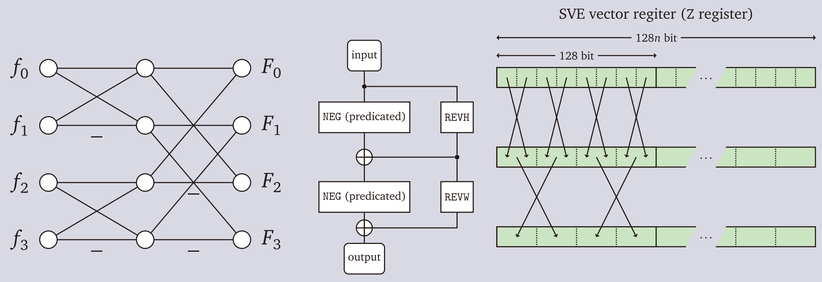

Figure 2. The Hadamard transform and its implementation in SVE

Sum of absolute transformed difference (SATD) is a primitive that is used in calculation of motion search and the most frequently used primitive. It consists of Hadamard transform of 4-elements and taking sum of all transformed elements. The original C implementation processes elements one-by-one, but our SVE implementation processes VL / 16 elements at once, where VL is vector length. The code uses REV{H,W} instructions to swap elements in the vector register.

Optimization example 2: interpolation filter

```
for (row = 0; row < height; row++) {
  for (col = 0; col < width; col++) {
    int sum = 0;
    for (i = 0; i < 8; i++)
      sum += src[col + i] * coeff[i];
    dst[col] = sum;
  }
  src += srcStride;
  dst += dstStride;
}
```

Figure 3. The interpolation filter primitive implementation in C (left), and in SVE (right)

Filter primitives are used to interpolate subpixels in a block. The filter can be processed as shown on the left: First loads data into a register, and copy the data to another 7 registers (Z1-Z7) using EXT instruction, with sliding the window. Multiply with coefficients, and finally accumulate all 8 registers element-wise and get the result. This way we can process VL / 8 elements at once. Gather-load and scatter-store instructions are used in vertical variants of the filter, which requires non-contiguous memory access.

Optimization example 3: Transform (DCT)

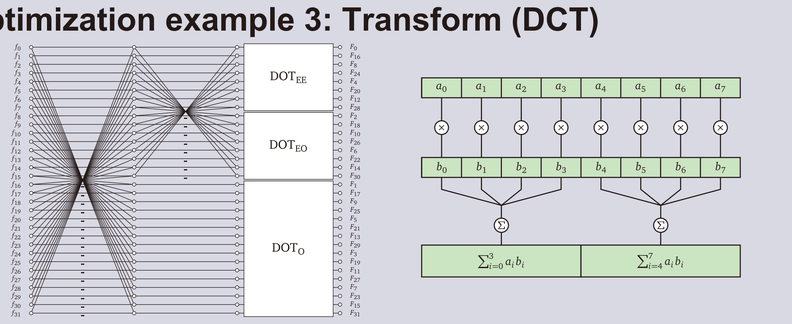

Figure 4. Signal flow diagram of “Partial Butterfly” algorithm in H.265/HEVC DCT (left), and the SVE’s SDOT instruction, which is used in the implementation (right).

H.265/HEVC uses a DCT-like transform to quantize information efficiently. The calculation is equivalent to 2 matrix-matrix multiplications. But thanks to the symmetries of the coefficient matrix, the number of multiplication can be reduced. This algorithm called “partial butterfly”. In the SVE implementation, using whole of vector, VL/64 line(s) are processed at once, resulting higher scalability. SVE’s SDOT instruction does most of calculations we need and contributes to the reduction of the number of instructions.

Evaluation

Scalability of primitives

The chart on the right (Figure 5) shows the ratio of the instruction count of SVE optimized primitives to that of non-SVE primitives, with varying the SVE vector length (VL). For all of the primitives shown in the picture, the number of instructions get reduced as the VL become longer. This indicates the program made full use of vector register, which VLA programming model aims. The only expception here is interp_vert_pp, whose instruction count increases at VL=2048. This is because our implementation processes at most single line (= 64 halfwords) at once, which is smaller than the VL. Processing multiple lines using whole vector may reduce instruction count for SVE machine with very long vectors. At VL=128, SVE and NEON (AArch64’s 128bit fixed-width SIMD) have the same vector length, but some SVE primitives (satd, sad, and dct32) still run in much smaller number of instructions. This is mainly because GCC failed to vectorize the code using NEON when SVE is turned off.

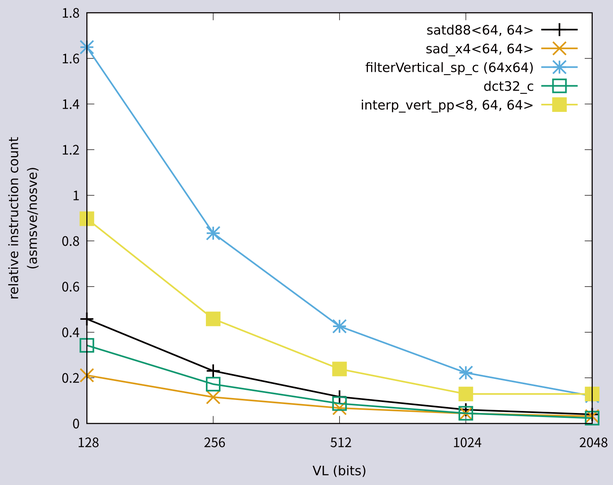

Figure 5. Number of instructions executed in optimized SVE primitives, compared with original implementation (no SVE instructions used, it uses only NEON)

Encoding a test clip

We used a short test clip to count the whole numbers of instructions executed in encoding. We compared our optimized version of x265 with the original source code, built without SVE enabled. Both binaries were built using GCC 9.2, and with optimization option -O3 used. The result (Figure 6) shows our implementation reduces up to 50% of instructions, compared with the non-SVE version. Also we observed that the number of executed instructions decrease as the vector gets longer. The figure also shows how much each function takes up of the whole instruction count. We observed that the number of executed instructions for most frequently used functions gets reduced, and that leads to the reduction of total number of instruction executed.

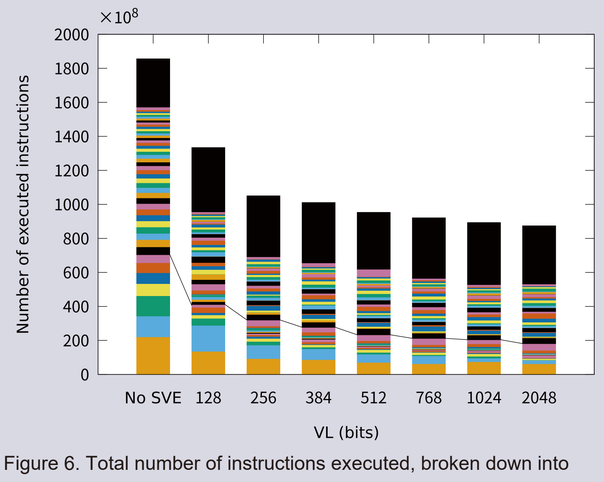

Figure 6. Total number of instructions executed, broken down into functions.

Conclusions & Future Work

We optimized x265 by re-implementing most heavily used functions using SVE instructions, and achieved reduction of about 50% of instructions executed using SVE instructions.

Future work

- Implement more primitives using SVE instructions
- Run on latency-aware simulators, like gem5
- Investigate SVE2 [3]

Acknowledgments

We would like to thank Mitsuhisa Sato, Jinpil Lee, and other members of Programming Environment Research Team of RIKEN Center for Computational Science for accepting the author as an intern and giving advice on writing and optimizing programs for SVE.

References

[1] Arm Limited. [n. d.]. Arm Instruction Emulator | Analyzing SVE programs – Arm Developer. https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-instruction-emulator/analyzing-sve-programs

[2] MulticoreWare Inc. [n. d.]. x265 HEVC Encoder / H.265 Video Codec. http://x265.org/

[3] N. Stephens. 2019. New Technologies in the Arm Architecture.

[4] N. Stephens et al. 2017. The ARM Scalable Vector Extension. IEEE Micro 37, 2 (March 2017), 26–39. https://doi.org/10.1109/MM.2017.35