Enhancement of convolution operation performance using SIMD of AArch64

Andrii Shevchenko[1,*,†] and Pylyp Prystavka[1,†]

[1] *National Aviation University, 1 Lubomyra Guzara ave., 03058 Kyiv, Ukraine*

Abstract

Optimization of two-dimensional convolution through 16-bit SIMD technologies of ARM x64 (aarch64) is considered. It is shown that by utilizing inline assembler and 16-bit SIMD commands of aarch64, one can achieve a significant performance increase compared to the similar functions of OpenCV. Throughout the research, filter coefficients were quantized to match the 8-bit range.

Keywords

convolution, SIMD, optimization, performance

1. Introduction

The process of automatic program code vectorization (APCV) is based on the SIMD instructions of the CPU. APCV is utilized in modern compilers, e.g., GCC and Clang/LLVM. The benefits that provide APCV can be achieved by compiling the program with -O3 (or "aggressive" -O4/-Ofast) flag (actually, the flag may differ depending on the platform and compiler). But as was shown in [1, 2], we achieve performance from APCV less than we need in the context of digital image (DI) processing. Moreover, DI problems have great importance due to the wide variety of applications in video-stream processing (stabilization, filtration, noise correction, or applying some effects for a single image, etc.). In processing DI, one should always take into account the following features:

1. The computational complexity of the method chosen.
2. Whether the method is optimized.
3. Hardware resources of the target architecture.

One can emphasize resource-demanding (but significant) operations of DI processing: convolution, scaling (mostly achieved through convolution), and analysis (of color, brightness, contrast, etc.). Convolution operation (CO) (1) is the simplest but most valuable and resource-demanding operation:

$$p_{i,j} = \sum_{k=0}^{r} \sum_{l=0}^{c} \Gamma_{k,l} P_{k+i-a,l+j-a'}, \tag{1}$$

where $i = a, \ldots, W - (r - a) - 1$, $j = a', \ldots, H - (c - a') - 1$ are indexing pixels of the destination image p; W and H are the width and height of the source P and destination p images (we neglect border effects in the destination at the moment), Γ is the kernel of the convolution (matrix $r \times c$), and a, a' is so-called "anchors" that define relative position of a filtered point within the kernel.

Equation (1) is rather general and perfectly compatible with cv::filter2D(...) function of the OpenCV library [3], but further we will give our attention to square kernels, i.e. $r = c$, and thus from now on we presume kernel to be square-shaped without special mentioning.

So, every pixel of the destination DI can be calculated simultaneously. This means that the task can be parallelized. To accomplish this task, hardware developers created a set of parallel computing platforms (PCP) (like Nvidia CUDA, ATI Stream Technology (ATI-ST), etc.) to perform these parallelizable problems. To create some common approach for all PCP Chronos is distributing the OpenCL API/lib (just like Nvidia distributes CUDA and so on). So every PCP developer provides the software toolkit to interact with the PCP: programming language with C-like syntax, additional modules, frameworks, etc.

Modern GPUs that can perform in parallel nearly any parallelizable task is the basis of PCPs. Currently, GeForce and ATI video accelerators are very popular for CNN learning and significantly accelerate the process. But the core/base of this calculation process is performed by shader blocks of GPU. The curious fact is that shader blocks of GPU are similar to mobile CPUs with RISK architecture, which are used in modern smartphones.

In conclusion, for to positive effect on the software that is using the DI processing (image filtration, scaling, edge detection, blurring, etc.), it is essential to provide speed improvement of CO. This will automatically lead to speed improvement in such fields/areas as multimedia (video codecs), CNN learning, etc. It is worth noting that CO (1) is the basis for (CNN) functioning if it (CO) is used in the base optimized computation approach like "Integer-Arithmetic-Only". Therefore, by accelerating CO we achieve higher CNN performance and decrease its learning time. Moreover, GPU architecture is based/has been created to improve/solve/speed up such tasks as CO and similar tasks.

CPITS-II 2024: Workshop on Cybersecurity Providing in Information and Telecommunication Systems II, October 26, 2024, Kyiv, Ukraine
[*] Corresponding author.
[†] These authors contributed equally.
lllandreyshevchenkolll@gmail.com (A. Shevchenko); chindakor37@gmail.com (P. Prystavka)

0000-0003-3863-0473 (A. Shevchenko); 0000-0002-0360-2459 (P. Prystavka)

In the current contribution, we propose a new method of CO optimization that utilizes ARMx64 SIMD-like aarch64 (NEON64) operations. Since NEON64 can be applied to integer-valued kernels, we will demonstrate a method for kernels with real values that allows this technique to be implied. Besides, to prove the suggested approach is effective we provide an experimental comparison of a human-made code (based on this approach) with recognized solutions like OpenCV lib.

The rest of the paper is organized as follows. First, we consider NEON64's pros and cons, and in subsection II-B we introduce the reduction/proposed method itself.

2. A brief overview of modern software optimization

Will perform the overview in a "bottom to top" style—consider hardware first, then software, and then algorithmic methods of performance enhancement.

2.1. Acceleration using hardware

It is worth noting that to significantly enhance software product performance, well-chosen hardware architecture is the most important. The first question is whether the task (e.g., CO) allows parallelization of data flow (or instructions flow). Flynn's taxonomy [4] gives a general perspective on possible solutions.

Today NEON64 (A64 instruction set; Neon SIMD for ARM64 CPUs) principles are implemented in both RISC (e.g., Cortex-A53-72/X1 ARM64 CPUs) [5] and CISC (e.g., Intel x64/x86 series) CPUs. To obtain access to such a feature (NEON64) specific extensions of the assembly language are needed. CISC architecture implies SSE_n and $AVX_{1/2}$ extensions of the assembly language. In contrast, MIMD principles are not implemented in modern CPUs but are partially supported by GPUs. As was noted before, based on RISC architecture modern GPUs provide parallel computing features due to shader blocks-CPU (SCPU). SCPU contains some specific 128/256/...-bit registers using whom partial MIMD principles are implemented. The number of these special registers is 32 or more, above the number of SIMD registers that modern ARM64 CPUs have. The bad part is you cannot access SIMD/MIMD instructions directly through the language that maintains the possibility of communicating with SIMD regs of SCPU. There are some preordered intrinsics and pre-implemented operations accessible: bit shifts, binary logic, etc. Mostly, programmers use specific frameworks to access the mentioned features, e.g. CUDA and OpenCL for GPU, or OpenCL for CPU/DSP/FPGA.

Except using GPU, one can employ co-processor units, e.g. Digital Signal Processor (DSP) like Qualcomm Hexagon. It has been developed for embedding into Snapdragon-6XX/8XX CPUs to reduce the CPU load by up to ~75% and improve audio/video encoding/decoding performance by up to ~18 times [6, 7]. Moreover, compared to simple NEON64, its performance is ~4 times higher. This DSP uses a very long instruction word (VLIW), which means multithreading at the assembler level (as SIMD) during one interruption, three assembly instructions with different inputs are processed.

2.2. Optimization using software

The software we use (e.g. compiler itself, additional libraries, frameworks) highly influences program performance (that we produce) by employing different optimizations to use more effective the hardware platform capabilities. In the scope of the current paper, we are primarily concerned with their ability to perform vectorization without significant loss in precision and speed.

Let's consider three well-known compilers: GNU Compiler Collection (GCC/G++) [8], Clang [9], and nvcc (compiles cu-files for CUDA).

The most popular nowadays is still the GCC compiler developed/supported by the FSF community. The first versions of GCC were a collection of compilers for different programming languages developed by Richard Stallman. Nowadays GCC is no longer a GNU C compiler now it is a GNU Compiler Collection. GNU is an optimizing compiler produced by the GNU Project supporting various programming languages, hardware architectures, and operating systems.

GCC's main competitor is Clang. For example, Apple already uses it as the basic compiler for its products. Moreover, the UNIX/BSD OS/distributives also use it as a default compiler. The Android NDK no longer uses GCC and by default, the clang compiler is used for it. Clang itself is a frontend for different programming languages, e.g. C, C++, Objective-C, Objective-C++, and OpenCL. The actual generation of binary code and vectorization is performed by the LLVM framework. Both GCC and Clang are performance-oriented, but still, they fail compared to human-made assembly code [1, 2, 10].

nvcc is the last compiler that we want to mention. It widely utilizes NVidia CUDA plus the power of C language, which significantly improves PC performance with NVidia GPU only. The main peculiarity is that these GPUs can use SIMT Architecture whose core feature is that the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads.

But as we can see, the mentioned compilers and technologies introduce significant heterogeneity in the field of program optimization. They represent a family of separated devices/technologies. In response, the OpenCL standard was developed (The Khronos Group Inc.) that is supported by all mentioned hardware developers and provides access to parallel computations on GPU/DSP/CPU.

But PCPs have a drawback—a big overhead on transferring data through the bus. To avoid the problem, programmers organize data into pools, which allows for achieving more than a 20-fold increase in performance compared to CPU (CNN learning perfectly fits in this model). But using big pools is not always the solution—while processing streams from a video camera does not at all.

One more reasonable approach to achieve performance enhancement of DI processing is supplied by different libraries (proprietary or not) like OpenCV or arm ComputeLibrary. Many of them contain NEON64-optimized code for armeaby-v7a and arm64-v8a. Another smart strategy is to use a collection of libraries that can be combined into a single framework. As a result, the advantages of one library compensate drawbacks of the others. OpenCV and ACL [9] are good examples of libraries comprising a wide variety of algorithms, including DI processing, and DI analysis. Moreover, OpenCV contains even modules for CNN learning, optimized for different CPU architectures that use SIMD (AVX$_{1/2}$/SSE$_4$, NEON64) and GPU optimized approaches/solutions. Also, OpenCV is well-known for its high-quality DI processing. Thus, further, we will OpenCV as a reference for comparison.

At the moment SIMD optimization has spread over a wide range of programming products, both proprietary and open-source. For example, the kernel of Windows 10 OS is widely used AVX$_{1/2}$/3DNow SIMD optimizations to achieve better performance (obviously, this influences the whole system). Oracle Java VM utilizes AVX$_{1/2}$/3DNow and thus any Java application runs faster. But, using SIMD optimization, they all face the issue of translating floating-point code to fixed-point with acceptable loss in precision. Therefore, it is quite complicated. Thus SIMD optimizations used in proprietary software are mostly non-disclosable.

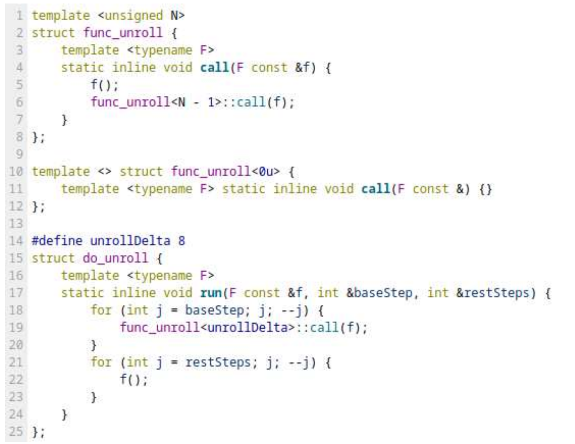
One more technique to mention is the so-called loop unrolling and tiling [11–13]. This technique avoids redundant comparison operations at the cost of slightly enlarging the out/binary file. It is mainly performed by utilizing the compiler or by introducing appropriate assembly inline code into the application.

Some libraries like ACL may use high-level programming language features (e.g., templates in C++) to perform loop unrolling. A simplified ACL-style code is provided in the listing to demonstrate an example implementation in Figure 1: Loop unrolling with C++ templates. Our previous paper [2] provided a detailed description that leads to a huge (over +25%) speed improvement to an algorithm. However, the ARM64 architecture was significantly improved compared to the ARMv7-A. If not go too deep in details main conclusion about this kind of approach is that we do not need this technique. Moreover, we have done some simple research in which we compare the speed of two equivalent functions, one with loop unrolling and another without it. The result was unexpected. The function with a loop unrolling gives a 3-5% speed reduction. To get proof about the fact that the loop was unrolled the IDA was used. As on the ARMv7-A arch, the cycle was unrolled on ARM64 by the clang (9 versions) compiler, and as expected the body of the bottleneck CO function part was repeated 8 times. But there is one thing to mention—the bottleneck CO function part was covered by redundant comparisons which can have such a negative effect. The ACL lib part that was optimized using NEON64 was rewritten without a loop unrolling approach.

This unusual fact gives food for think and in further research about different CO approaches/methods, we will cover (go deeper) them.

2.3. Optimization using special algorithms

Let's focus on CO. The primary obstacle for SIMD optimization is the act of translation of floating-point CO algo into fixed-point algorithm CO algo with an acceptable loss of precision or even without it. First of all, SIMD operations will be performed on integers further.



```
template <unsigned N>
struct func_unroll {
    template <typename F>
    static inline void call(F const &f) {
        f();
        func_unroll<N - 1>::call(f);
    }
};

template <> struct func_unroll<0u> {
    template <typename F> static inline void call(F const &) {}
};

#define unrollDelta 8
struct do_unroll {
    template <typename F>
    static inline void run(F const &f, int &baseStep, int &restSteps) {
        for (int j = baseStep; j; --j) {
            func_unroll<unrollDelta>::call(f);
        }
        for (int j = restSteps; j; --j) {
            f();
        }
    }
};
```

Figure 1: Loop unrolling with C++ templates

Thus, we should represent elements of the kernel Γ from (1) in a suitable form:

$$\Gamma_{i,j} = \nu\gamma_{i,j}, \nu \in R, \gamma_{i,j} \in Z \qquad (2)$$

where ν is a coefficient for normalization. Now we can perform/discuss the most resource-demanding part (additions and multiplications) in a SIMD style and afterward normalize the result.

Any kernel can be represented in form (2), but the more precise the result we want, the more digits should have $\gamma_{i,j}$. So, we should set some constraints on γ to avoid overflow when doing CO because of the platform's limitations on which we intend to run the program.

Suppose, every pixel in the original image is represented as a byte and thus possesses 8-bit values 0, …, 255. The same range is possessed by kernel elements $\gamma_{i,j}$. Intermediate results are stored as 16-bit signed or unsigned values. To warrant that no overflow occurs, we should ensure that it does not happen on any algorithm step. If the kernel has positive elements only, a condition we need looks as follows

$$(2^8 - 1) \times \sum_{i=0}^{r} \sum_{j=0}^{r} \gamma_{i,j} \leq 2^{16} - 1. \qquad (3)$$

Substantially, this means that even the largest possible inputs from the image do not lead to overflow.

If the kernel contains negative elements, the condition should be much more complicated and depend on the order of additions when doing CO. Instead, we will use much stronger but more straightforward to check the condition

$$(2^8 - 1) \times \sum_{i=0}^{r} \sum_{j=0}^{r} |\gamma_{i,j}| \leq 2^{16-1} - 1, \qquad (4)$$

independent of the operations' order. Moreover, this condition can be slightly relaxed—we can use it for positive and negative entries of the kernel γ separately. And the last thing to mention: one can easily obtain similar results for signed/unsigned 32-bit intermediate values by substituting 16 ⟶ 32 in (3) and (4).

What we propose is selecting for giving Γ the most extensive ν possible, such that γ still satisfies (3) or (4) (which one depends on whether the kernel is purely positive or not). Of course, we shouldn't be concerned about whether any valuable kernels can be reduced to a suitable form/size because there are plenty of them.

In conclusion, modern hardware provides mechanisms for vectorization, i.e., SIMD technologies, that programmers can use to enhance the performance of the application. In most cases, this technology is utilized by the compiler to generate binary code without the participation of the programmer. A suitable choice of the library may be handy as well—many libraries contain SIMD-optimized code. But in some cases, human intervention is needed to get the most optimal result. More specifically—the code must represent the function/code which can be/suitable for the SIMD optimization. However, it is not always possible, and in our case restrictions (3,4) should be satisfied. In the next section, we will provide a new method of CO optimization and then compare it with existing results from OpenCV lib.

3. Optimization of Convolution Operation using SIMD

In the current contribution, we propose a new Convolution Operation (CO) optimization method based on the SIMD technique. We presume that the target kernel satisfies the 3rd condition. This section will provide all necessary considerations and an inline assembly code that illustrates the proposed approach. The following section will be devoted to an experimental comparison of this method's performance to known CO implementations of OpenCV.

Regarding condition (4), the provided code should be just slightly modified. Therefore, we will avoid redundant code listings and deliver code that realizes condition (3). In contrast, all necessary modifications for a realization of condition (4) will be described at the end of the section. We start with the basic implementation of CO (see Figure 2b). It contains no specific optimizations but still is a good point to begin our considerations.

Here v_n are the NEON64 registers. Regarding syntax and instructions order, we will strictly follow ARM reference manuals. For the sake of simplicity, we avoided normalization by the coefficient v in (see Figure 2b), but for completeness, let us provide it separately (see Figure 2c).

In (Figure 2c) we suppose data for normalization to be stored in registers v12–v15, while v1[0] contains the normalization coefficient v. The presented code is in some sense multipurpose and may be used with different CO implementations.

Now we switch gears to the CO optimization itself by utilizing NEON64. In (see Figure 2b) have been provided a naive version/approach of this operation (in assembly code). But this variant contains one significant drawback—data loading. The data loading/storing process is the slowest operation because it involves sub/inner processes like communication with the CPU and RAM. Even though such hardware approaches like CPU cache cover this operation, it is still slow.

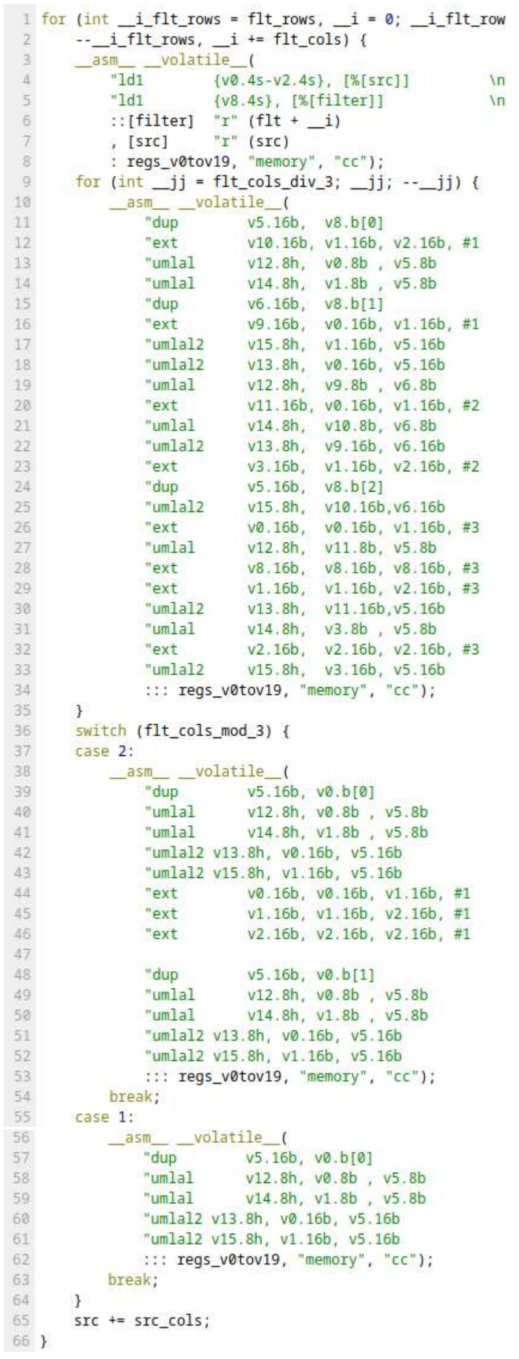
To avoid this problem, one of the registers was used as a buffer. The following approach (see Figure 2a) avoids this problem by using one of the registers as a buffer. It is known that simultaneous loading of 16 bytes is quicker than loading them one by one. Thus we use one register for preloading extra data and then use this data to perform byte-by-byte shift to exclude redundant load operations.

Let's comment on the sections of this code/approach (Figure 2b). This is a naïve approach representing the loading operation for each kernel element and loading source image elements (lines 5, 6). The loading and storing operations are the most expensive operations. (lines from 8–11) represent the multiply-and-accumulated image values (v2, v3) with the kernel element (v0). The results of these operations are stored in the buffer regs (v12, v13, v14, v15). The buffer regs represent the result of the 8-bit multiplication of image values on each kernel element extended up to 16-bit unsigned int using the "umlal" operation. These operations are performed for every kernel element. So as you can see, this is time time-consuming approach.

Let's comment on the sections of this code/approach (see Figure 2a): line 4 loading 48 bytes of grayscale image to v0–v2; line 5 loading 16 bytes of CO kernel in v8; lines 13,14,17,18 provide conversion from 8-bit to 16-bit and multiplication calculation with kernel element in v5 simultaneously. Please note that v0–v2 registers contain part of the image that should be convolved with the kernel stored in v8. Register v2 is exploited as a buffer for 16 more bytes of the input image to speed up the CO by utilizing the "ext" operations. Moreover, data from buffer v2 is being used to perform cyclically shifting content of v0 (line 26), v1 (line 29), and v2 (line 32 with itself) byte-by-byte performed with the "ext" command. It is not quite clear but we utilize different names of the registers to save shifted states of v0–v2 (lines 12, 16, 11, 23) which is called the register rename technique. Also as you can see we utilized some reordering of instructions which brought little obfuscation. Nevertheless, all this gives about 7-10% speedup in comparison to the ordered instruction set which utilizes the process of saving all the time in the same names registers names v0..v2.

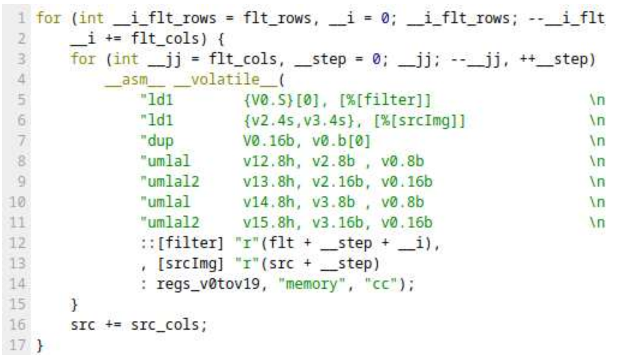
Moreover, if we save the result of the shift in the same register name (like v0, v1, v2), we receive speed-reduced impacts. This is because the operation "ext" saves the result in a state of progress, and when the next operation tries to obtain the content of the v0 (or v1, or v2), it produces the waiting/bottleneck state. So, the more such conditions appear in the program, the less win of time provided by the algorithm. The most resource part of optimized CO algo (Fig. 2a) was almost entirely described by us. Finally, the "case" state (Fig. 2a, lines 37 up to 63) represents the calculation finishing of the kernel row.

So as you can see the main feature of the presented approach (see Figure 2a) is the usage of cyclic shift (i.e., ext v10.16b, v0.16b, v1.16b, #1) that provides the kernel buffering, and thus, we need fewer operations of loading. One more thing that should be mentioned is the pre-save of the shifted data (see Figure 2a) (in lines 11,15) on to 1 element and (in lines 19, 22) on to 2 elements were used for the current iteration of CO. Other "ext" operations (lines 25, 28, 31) provide the data initialization for the next iteration of CO. Worth noting, that provided (see Figure 2a) demands a kernel containing not more than 16 elements in one row. Another variation of this interpretation in which CO kernel size is more than 16 elements should utilize data reinitialization of the base registers, which can be seen (in lines 3–4).



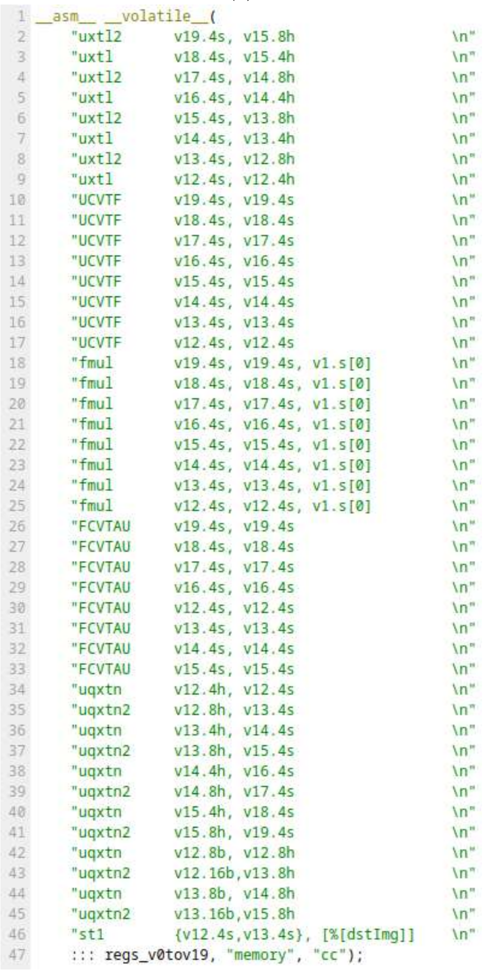
```
for (int __i_flt_rows = flt_rows, __i = 0; __i_flt_row
    --__i_flt_rows, __i += flt_cols) {
    __asm__ __volatile__(
        "ld1        {v0.4s-v2.4s}, [%[src]]        \n
        "ld1        {v8.4s}, [%[filter]]           \n
        ::[filter]  "r" (flt + __i)
        , [src]     "r" (src)
        : regs_v0tov19, "memory", "cc");
    for (int __jj = flt_cols_div_3; __jj; --__jj) {
        __asm__ __volatile__(
            "dup        v5.16b,  v8.b[0]
            "ext        v10.16b, v1.16b, v2.16b, #1
            "umlal      v12.8h,  v0.8b , v5.8b
            "umlal      v14.8h,  v1.8b , v5.8b
            "dup        v6.16b,  v8.b[1]
            "ext        v9.16b,  v0.16b, v1.16b, #1
            "umlal2     v15.8h,  v1.16b, v5.16b
            "umlal2     v13.8h,  v0.16b, v5.16b
            "umlal      v12.8h,  v9.8b , v6.8b
            "ext        v11.16b, v0.16b, v1.16b, #2
            "umlal      v14.8h,  v10.8b, v6.8b
            "umlal2     v13.8h,  v9.16b, v6.16b
            "ext        v3.16b,  v1.16b, v2.16b, #2
            "dup        v5.16b,  v8.b[2]
            "umlal2     v15.8h,  v10.16b,v6.16b
            "ext        v0.16b,  v0.16b, v1.16b, #3
            "umlal      v12.8h,  v11.8b, v5.8b
            "ext        v8.16b,  v8.16b, v8.16b, #3
            "ext        v1.16b,  v1.16b, v2.16b, #3
            "umlal2     v13.8h,  v11.16b,v5.16b
            "umlal      v14.8h,  v3.8b , v5.8b
            "ext        v2.16b,  v2.16b, v2.16b, #3
            "umlal2     v15.8h,  v3.16b, v5.16b
            ::: regs_v0tov19, "memory", "cc");
    }
    switch (flt_cols_mod_3) {
    case 2:
        __asm__ __volatile__(
            "dup        v5.16b, v0.b[0]
            "umlal      v12.8h, v0.8b , v5.8b
            "umlal      v14.8h, v1.8b , v5.8b
            "umlal2 v13.8h, v0.16b, v5.16b
            "umlal2 v15.8h, v1.16b, v5.16b
            "ext        v0.16b, v0.16b, v1.16b, #1
            "ext        v1.16b, v1.16b, v2.16b, #1
            "ext        v2.16b, v2.16b, v2.16b, #1

            "dup        v5.16b, v0.b[1]
            "umlal      v12.8h, v0.8b , v5.8b
            "umlal      v14.8h, v1.8b , v5.8b
            "umlal2 v13.8h, v0.16b, v5.16b
            "umlal2 v15.8h, v1.16b, v5.16b
            ::: regs_v0tov19, "memory", "cc");
        break;
    case 1:
        __asm__ __volatile__(
            "dup        v5.16b, v0.b[0]
            "umlal      v12.8h, v0.8b , v5.8b
            "umlal      v14.8h, v1.8b , v5.8b
            "umlal2 v13.8h, v0.16b, v5.16b
            "umlal2 v15.8h, v1.16b, v5.16b
            ::: regs_v0tov19, "memory", "cc");
        break;
    }
    src += src_cols;
}
```

(a)



```
for (int __i_flt_rows = flt_rows, __i = 0; __i_flt_rows; --__i_flt
    __i += flt_cols) {
    for (int __jj = flt_cols, __step = 0; __jj; --__jj, ++__step)
        __asm__ __volatile__(
            "ld1        {V0.S}[0], [%[filter]]                  \n
            "ld1        {v2.4s,v3.4s}, [%[srcImg]]              \n
            "dup        V0.16b, v0.b[0]                         \n
            "umlal      v12.8h, v2.8b , v0.8b                   \n
            "umlal2     v13.8h, v2.16b, v0.16b                  \n
            "umlal      v14.8h, v3.8b , v0.8b                   \n
            "umlal2     v15.8h, v3.16b, v0.16b                  \n
            ::[filter] "r"(flt + __step + __i),
            , [srcImg] "r"(src + __step)
            : regs_v0tov19, "memory", "cc");
    }
    src += src_cols;
}
```

(b)



```
 1 __asm__ __volatile__(
 2     "uxtl2      v19.4s, v15.8h                      \n"
 3     "uxtl       v18.4s, v15.4h                      \n"
 4     "uxtl2      v17.4s, v14.8h                      \n"
 5     "uxtl       v16.4s, v14.4h                      \n"
 6     "uxtl2      v15.4s, v13.8h                      \n"
 7     "uxtl       v14.4s, v13.4h                      \n"
 8     "uxtl2      v13.4s, v12.8h                      \n"
 9     "uxtl       v12.4s, v12.4h                      \n"
10     "UCVTF      v19.4s, v19.4s                      \n"
11     "UCVTF      v18.4s, v18.4s                      \n"
12     "UCVTF      v17.4s, v17.4s                      \n"
13     "UCVTF      v16.4s, v16.4s                      \n"
14     "UCVTF      v15.4s, v15.4s                      \n"
15     "UCVTF      v14.4s, v14.4s                      \n"
16     "UCVTF      v13.4s, v13.4s                      \n"
17     "UCVTF      v12.4s, v12.4s                      \n"
18     "fmul       v19.4s, v19.4s, v1.s[0]             \n"
19     "fmul       v18.4s, v18.4s, v1.s[0]             \n"
20     "fmul       v17.4s, v17.4s, v1.s[0]             \n"
21     "fmul       v16.4s, v16.4s, v1.s[0]             \n"
22     "fmul       v15.4s, v15.4s, v1.s[0]             \n"
23     "fmul       v14.4s, v14.4s, v1.s[0]             \n"
24     "fmul       v13.4s, v13.4s, v1.s[0]             \n"
25     "fmul       v12.4s, v12.4s, v1.s[0]             \n"
26     "FCVTAU     v19.4s, v19.4s                      \n"
27     "FCVTAU     v18.4s, v18.4s                      \n"
28     "FCVTAU     v17.4s, v17.4s                      \n"
29     "FCVTAU     v16.4s, v16.4s                      \n"
30     "FCVTAU     v12.4s, v12.4s                      \n"
31     "FCVTAU     v13.4s, v13.4s                      \n"
32     "FCVTAU     v14.4s, v14.4s                      \n"
33     "FCVTAU     v15.4s, v15.4s                      \n"
34     "uqxtn      v12.4h, v12.4s                      \n"
35     "uqxtn2     v12.8h, v13.4s                      \n"
36     "uqxtn      v13.4h, v14.4s                      \n"
37     "uqxtn2     v13.8h, v15.4s                      \n"
38     "uqxtn      v14.4h, v16.4s                      \n"
39     "uqxtn2     v14.8h, v17.4s                      \n"
40     "uqxtn      v15.4h, v18.4s                      \n"
41     "uqxtn2     v15.8h, v19.4s                      \n"
42     "uqxtn      v12.8b, v12.8h                      \n"
43     "uqxtn2     v12.16b,v13.8h                      \n"
44     "uqxtn      v13.8b, v14.8h                      \n"
45     "uqxtn2     v13.16b,v15.8h                      \n"
46     "st1        {v12.4s,v13.4s}, [%[dstImg]]        \n"
47     ::: regs_v0tov19, "memory", "cc");
```

(c)

Figure 2: CO optimization with SIMD NEON64: (a) optimized approach; (b) naive approach; (c) normalization procedure and saving the result.

Let's comment on the sections of this code/approach (Figure 2c). This section provides the normalization of coefficient ν. Lines from 2–9 represent the conversion process from 16-bit data types up to 32-bit data types. Saving all data needs twice as much register stack (v12–v19) as it was before (v12–v15). Lines from 10–17 represent the data conversion from unsigned integer (32-bit) up to (32-bit) floating-point. Finally, lines 19–25 describe the normalization process with the ν coefficient (placed in v1.s[0]). All other lines (26–46) represent the reverse process: the normalized data converts from the (32-bit) floating-point up to (8-bit) unsigned integer (lines 26-45) and the result saving (line 46).

As we mentioned earlier, this code works for kernels satisfying conditions (3). To make it applicable to kernels satisfying (4), we need to change all "umlal" operations to "smlal" but before it, the extended operation is required (like "sxtl"). These small but crucial changes transform (see Figure 2a) into code that works with signed integer kernels. Depending on elements in the given kernel, one can choose between these two options.

In conclusion, we found a class of kernels that allow significant optimization CO utilizing NEON64 and implementing appropriate code/algo. For example, the Subband low pass filtering kernel, like (5).

$$\gamma = \begin{pmatrix} 1 & 6 & 1 \\ 6 & 36 & 6 \\ 1 & 6 & 1 \end{pmatrix}, \ \nu = \frac{1}{64} \qquad (5)$$

Furthermore, we achieved a significant CO speedup by exploiting substantial differences in time for simultaneous 16-byte loading with a byte-shift approach compared to one-by-one line loading. More detailed results and considerations of the measurement procedure will be presented in the following section.

4. Experimental setup and results

Ground truth. To evaluate our results certain reference is needed. As the etalon, we chose functions cv::filter2d(...) from the OpenCV library. The latter is well-known among AI and DIP researchers due to its high-quality and optimized code. Especially when quick prototyping is needed.

For comparison, we used the latest stable tag available when we started to research. The release tag is 4.5.2 (2021-04-02 11:23) for OpenCV. The compilation was performed with clang-9 - the latest stable clang version. We ensured that libraries utilize vectorization, compiling them with flags: -DCMAKE_BUILD_TYPE=RELEASE -DENABLE_NEON=ON ... and the compilation process was with the verbose mode on. The result is that some critical fields like "CPU_BASELINE" (NEON F16) and "C++ flags (Release)" (...-O3 -DNDEBUG...) provided needed content. Also, we mention the fact that OpenCV lib was linked as a dynamic library.

Devices. To make our measurements more relevant, we used such a device as Odroid-C4. This helps us understand the influence of architecture, CPU series, and other parameters on the execution time. The Odroid-C4 CPU is Cortex-A55; the OS is Ubuntu 20.04; Linux 5.7.0-odroid-arm64 is the kernel, and its API is aarch64. The CPU series of this device is Amlogic S905X3 which is more powerful than the latest Raspberry Pi CPUs.

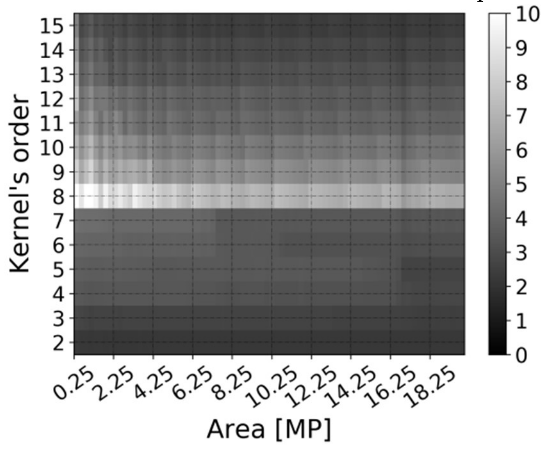
Measurement procedure. The pivoting parameter we need to measure is the execution time of each function. Such measurement might be tricky since it is highly susceptible to transition processes in any GNU OS (Ubuntu, Android, etc.).

To avoid this problem, we used the following procedure: each function (cv::filter2d(...) and proposed method - newCO(...)) was successively called three times (for robustness and to simulate Grayscale processing), and the result was stored to the array—this is one data point. Then, after collecting 35 data points, we calculated the median value and treated it as twice the function's execution time under consideration.

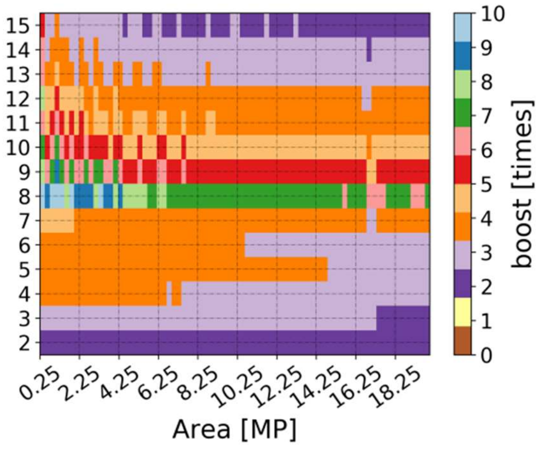
Kernel sizes varied 2×2, 3×3, ..., 15×15 for experiments with our implementation and cv::filter2d(...). DIs were generated with equal width and height, the corresponding formula follows

$$W_{\text{image}} = H_{\text{image}} = \left[\frac{125\sqrt{n}}{8}\right] \times 32 + W_{\text{kernel}} - 1,$$

where square brackets [...] denote the integer part of the number. Results are further presented in the form of fractions cv::filter2d(...) execution time divided by execution time of proposed/our implementation.



(a)



(b)

Figure 3: Performance comparison of the CO usage of cv::filter2D vs. the proposed method on the devices with Cortex-A55 ARM CPU.

Color intensity designates relative time consumption for reference function about the proposed method. Acceleration one may achieve by using the presented approach instead of the reference function (the brighter is color—the greater is acceleration). Legends on each plot designate how to translate color to acceleration; if this number is greater than 1, it is profitable to use the proposed method.

5. Results

First, we compared the time consumption of the proposed code (see Figure 2a) and reference function cv::filter2d(...). The result is presented in Figs. 5a and 5b. As coordinates, we use sizes of kernel and image. At the same time, color intensity designates acceleration, which one may achieve using the proposed method instead of the reference method (e.g., a fraction of the execution times of the reference function divided by the execution time of the proposed method).

Despite the presented results demonstrating the advantage of the proposed method, there is still room for improvement. For example, it seems the compiler cannot unroll cycles effectively on its own, and we mentioned this above. But if we do the same as was done in ACL—unroll all "bottle-neck cycles" on our own, it seems we can achieve a more speedy approach/results. Thus, we may reach an additional 10-20% acceleration by utilizing techniques [11–13] by writing cycle unrolling with the online assembly by hand.

Results for the modified code are shown in Figure 3. We have compared the time consumption of the proposed method (see Figure 2a) and function cv::filter2d(...). Besides, we varied image sizes up to 4500×4500 (~20 [MP]) to emulate modern cameras and picture libs.

As Figure 3 suggests, acceleration is independent (almost) of the input size, e.g. complexity (big-O) of our solution and reference solutions coincide. Some small decline in acceleration (but it is still greater than 2) may be noted for big kernels (13×13 … 15×15) and smaller kernels (2×2 … 4×4). Regarding mean acceleration, it is estimated as approximately 3.7 times.

It is worth noting that we didn't use parallelism for acceleration. Moreover, no preprocessing, e.g., image tiling, was performed. Probably, this technique may increase the performance of the approach as well.

6. Conclusions

In conclusion, we propose a method of convolution operation acceleration. We have shown that speed improvement can be achieved if kernels have been reduced to integer values that allow SIMD command usage. Furthermore, despite SIMD itself leading to a significant boost of performance, we were able to push the frontiers even further by exploiting the considerable difference in time for simultaneous 32-byte loading compared to their one-by-one loading and using buffer (one-time load for the kernel row—48-byte), and loading operations are partially substituted with cyclic shift.

About ALC, we should mention in addition. There was a severe code rewriting event in this lib. Furthermore, the patches became cumulative ("less description more code"). This fact brought more obscurity/obfuscation than clarity/understanding. So, we will compare the ACL lib and modifications of our suggested approach in our following paper but it is needed to mention that ALC provides all additional code optimization approaches that we mentioned above (cycle unrolling, image tiling, etc.).

To test the approach we performed a comparison with the cv::filter2D(...) function from the OpenCV library. Our results suggest the current approach leads to significant speedup (mean values: ~3.7× compared to OpenCV). Measuring acceleration for different kernels and images we observed no dependence on image size, but kernel size may influence the result—for kernels smaller than 8×8 we were able to achieve ×7.379 acceleration compared to cv::filter2D(...), while for larger kernels presented approach allows ~3.7 speedup.

We expect the current approach to be useful for real-time image processing and convolutional neural network training as it significantly reduces processing time.

References

[1] P. Prystavka, A. Shevchenko, Investigation of the Implementation of the Linear Operator of Digital Image Convolution in 16-bit Computing, Actual Problems of Automation and Information Technologies, no. 20 (2016) 78-90.

[2] A. Shevchenko, V. Tymchyshyn, A SIMD-based Approach to the Enhancement of Convolution Operation Performance, CMiGIN (2019).

[3] Image Filtering (2021). URL: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04.

[4] M. J. Flynn, Very High-Speed Computing Systems, in: Proceedings of the IEEE 54.12 (1966) 1901–1909.

[5] ARM Cortex- A53 MPCore Processor Technical Reference Manual (2021). URL: https://developer.arm.com/documentation/ddi0500/j.

[6] Qualcomm Extends Hexagon DSP (2013) URL: http://pages.cs.wisc.edu/~danav/pubs/qcom/hexagon_microreport2013_v5.pdf.

[7] Qualcomm Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications (2013). URL: https://developer.qualcomm.com/download/hexagon/hexagon-dsp-architecture.pdf.

[8] Griffith, Arthur. GCC: The Complete Reference. McGraw-Hill, Inc., 2002.

[9] B. C. Lopes, R. Auler. Getting Started with LLVM Core Libraries. Packt. Publishing ltd. (2014).

[10] ARM Compute Library (2021). URL: https://developer.arm.com/ip-products/processors/machine-learning/compute-librar

[11] A. Nicolau, Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation. Cornell University (1985).

[12] Xue, Jingling. Loop Tiling for Parallelism, vol. 575 (2000).

[13] T. Veldhuizen, Expression Templates. C++ Report 7.5 (1995) 26–31.