CEUR-WS.org/Vol-2588/paper37.pdf

A SIMD-based Approach to the Enhancement of Convolution Operation Performance

Andrii Shevchenko [1] [0000-0003-3863-0473] and Vitaly Tymchyshyn [2] [0000-0003-2292-600X]

[1] National Aviation University, Kyiv, Ukraine
[2] Bogolyubov Institute for Theoretical Physics, Kyiv, Ukraine
lllandreyshevchenkolll@gmail.com, yu.binkukoku@gmail.com

Abstract. Optimization of two-dimensional convolution by means of 16-bit SIMD technologies (ARM-NEON) is considered. It is shown that utilizing 16-bit SIMD NEON and built-in assembler one can achieve a significant increase in performance compared to the similar functions of OpenCV, ACL, and ARM Compute library. Throughout the research, ltercoefficients were quantized to match the 8-bit range.

Keywords: Convolution, vectorization, SIMD, optimization, CPU.

1 Introduction

The process of automatic program code vectorization is based on the SIMD instructions of CPU. It is utilized in modern compilers, e.g. GCC and Clang/LLVM. One can achieve automatic optimization/vectorization compiling the program with -O3 (or "aggressive" -O4/-Ofast) flag (actually, flag may differ depending on the platform and compiler). But as was shown in [1], we often need higher performance for digital image (DI) processing than we achieve automatically. Moreover, DI problems are of great importance due to the wide variety of applications in video-stream processing (stabilization, filtration, noise correction, etc.) and creating different effects for a single image. Processing DI, one should always take into account the following features:

1) computational complexity of the method chosen;
2) whether the method is optimized;
3) hardware resources of the target architecture.

One of the possible ways of computational complexity decrease is to develop a new particular method for a particular task, e.g. as in [2].

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 valuable and resource-demanding operation:

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

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0) CMiGIN-2019: International Workshop on Conflict Management in Global Information Networks.

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 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' - 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 OpenCV library [3] and ARM Computer Library (ACL), but further we will give our attention to square kernels, i.e. $c = r$, and thus from now on we presume kernel to be square-shaped without special mentioning.

The good part is that every pixel of the destination, at least in principle, can be calculated simultaneously - the task is parallelizable. Parallelizable problems are of such importance that hardware developers created a set of parallel computing platforms (PCP): Nvidia CUDA, ATI Stream Technology (ATI-ST), etc., all accessible through OpenCL API. Every PCP developer provides software toolkit to interact with the PCP: programming language with C-like syntax, additional modules, frameworks, etc. Basis for PCPs are modern GPUs that are able to perform in parallel nearly any parallellizable task. For example, at the moment GeForce GTX 1080 Ti is very popular for CNN learning and significantly accelerates the process. One should notice, shader blocks of GPU are similar to mobile CPUs with RISK architecture that are used in modern smartphones.

In conclusion, improvement of CO performance positively affects nearly any software on any platform due to the wide variety of tasks it is involved in: DI filtration (sharpening, edge detection, blurring, etc.), DI scaling, CNN learning, multimedia, etc. It is worth noting, that CO (1) is the basis for convolutional neural networks (CNN) functioning. Therefore, accelerating CO we achieve higher CNN performance and decrease its learning time. Moreover, in some sense CO and similar tasks have shaped modern approaches to GPU architecture, that emphasizes their importance.

In current contribution we propose a new method of CO optimization that utilizes SIMD. Since SIMD can be applied to integer-valued kernels only, we will demonstrate a reduction method for real-valued kernels that allows this technique to be implied. Besides, we provide experimental comparison of a human-made code based on this approach with another recognized solutions (OpenCV library, AMD ACL, and ARM Compute library). The rest of the paper is organized as follows. First, we consider SIMD pros and cons and in subsection II-B we introduce the reduction method itself.

2 A brief overview of modern software optimization

We will perform the overview in a "bottom to top" style - we consider hardware first, then software, and then algorithmic methods of performance enhancement.

2.1 Acceleration by means of hardware

Well chosen hardware architecture may significantly enhance software product performance. A general perspective on possible options is given by the Flynn's taxono-

my [4] and the first question to answer is whether the task (e.g. CO) allows parallelization of data flow or instructions flow. Today SIMD principles are implemented in both RISC (e.g. Cortex-A8-23 and Cortex-A53-72 ARM CPUs) [5] and CISC (e.g. Intel x86 series) CPUs. One can access this feature with specific extensions of assembly language - NEON for RISC architecture, while CISC architecture implies SSE_n and $AVX_{1/2}$ usage. In contrast, MIMD principles are not implemented in modern CPUs, but partially supported by GPUs.

Modern GPUs provide parallel computing features by means of shader blocks-CPU (SCPU), based on RISC architecture, that are used in parallel. Principles of MIMD are achieved by SIMD/MIMD-like instructions for SCPU --- vector instructions for numerous 128/256/...-bit registers (there are 32 or more registers, that is above the number that modern CPUs have). The bad part is you cannot access SIMD/MIMD instructions directly only a small number of intrinsics and pre-implemented operations are accessible: bit shifts, binary logic, etc. Most cases programmers use specific frameworks to access mentioned features, e.g. CUDA and OpenCl for GPU, or OpenCl for CPU/DSP/FPGU. Prevalence of mentioned frameworks led to most GPU manufacturers get certified by AMD/ATI (compatibility with OpenCl) or Intel/NVidia (compatibility with CUDA).

Except using GPU, one can employ co-processor units, e.g. Digital Signal Processor (DSP). For example, Qualcom has developed DSP-Hexagon for embedding into Snapdragon-625/635/835/825 CPUs. This DSP provides very long instruction word (VLIW), which means multithreading at the assembler level - during one interruption 3 assembly instructions with different inputs are processed. Compared to simple SIMD (NEON32 or NEON64) its performance is 4 times higher. Algorithms, optimized for DSP, reduce CPU load up to $\sim$ 75% and improve audio/video encoding/decoding performance up to $\sim$ 18 times [6], [7].

2.2 Optimization by means of software

Software we use to produce binary code (e.g. compiler itself, additional libraries, frameworks) highly influences program performance by employing different optimizations and hardware platform capabilities. In scope of current article we are mostly concerned with their ability to perform vectorization without significant loss in precision. Let's consider three well-known compilers: GNU Compiler Collection (GCC/G++) [8], Clang [9], and nvcc (compiles cu-files for CUDA).

Probably, the most popular nowadays is GCC developed and supported by FSF community. Actually, GCC, first developed by Richard Stallman, is a whole collection of compilers suitable for different programming languages and architectures. Its main competitor is the ``rising star'' of compilers - Clang. For example, Apple already uses it as the basic compiler for its products. 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 (see comparison [1] Clang ndk-r14b vs Inline Assembly on Android 5.5.1 (x64) phone with CPU-MT6752).

The last compiler we want to mention is nvcc. It utilizes CUDA and thus allows significant improvement of performance on platforms with NVidia GPU. But as we can see, mentioned compilers and technologies introduce large heterogeneity to the field of program optimization. In a responce 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.

Except all the advantages of PCPs, they have a drawback - big overhead on transferring data. To avoid the problem, programmers organize data into pools 100 ~ 200, that allows to achieve 20-fold increase in performance compared to CPU. But using big pools is not always the solution - while CNN learning perfectly fits in this model, processing stream from video-camera does not at all.

Except for good choice of compiler, one can achieve performance enhancement using optimized binary code of frequently used functions like CO, scaling, etc. supplied by different libraries. Many of them contain SIMD-optimized code for armeaby-v7a and arm64-v8a. Besides, a collection of libraries can be combined into a single framework in such a way, that advantages of one library compensate drawbacks of the others. OpenCV and ACL [10], [11] are good examples of libraries comprising a wide variety of algorithms, including DI processing, DI analysis, and even module for CNN learning, that are optimized for different CPU architectures and their SIMD: $\mathrm{AVX}_{1/2}$, $\mathrm{SSE}_{4.4}$, $\mathrm{ARM\ NEON}_{x32/x64}$. OpenCV is well-known and of a high quality, but ACL has better extensibility due to modular architecture and seems to perform better on CO-like tasks (e.g. it is up to ×14 times faster compared to OpenCV on CO for CNN in one thread [10], [11]). Thus, further we will use both of them as a reference for comparison.

At the moment SIMD optimization has spread over the wide range of programming products, both proprietary and open-source. For example, kernel of Windows 10 uses $\mathrm{AVX}_{1/2}$ to achieve better performance (obviously, this influences the whole system), while Oracle Java VM utilizes $\mathrm{AVX}_{1/2}$/3DNow and thus any Java application runs faster. Game engines of id Tech 2-4 (e.g. the one used in Quake III Arena) are good example of open-source projects with SIMD optimization. But, using SIMD, they all face the issue of translating floating-point code to fixed-point with acceptable loss in precision. This can be quite complicated, thus SIMD optimizations used in proprietary software are mostly non disclosable.

One more technique to mention is so-called loop unrolling and tiling [12], [13], [14], [15] that allows avoidance of redundant comparison operations (e.g. <,>) in cost of slightly enlargening the code. It is mostly performed by means of compiler or by introducing appropriate assembly inline-code into the application. Some libraries like ACL may take advantage of high-level programming language features (e.g. templates in C++) to perform loop unrolling. A simplified ACL-style code is provided in listing to demonstrate example implementation (see **Помилка! Джерело посилання не знайдено.**).

```
1 template <unsigned N>
2 struct func_unroll {
3     template <typename F>
4     static inline void call(F const& f) {
5         f();
6         func_unroll<N - 1>::call(f);
7     }
8 };
9
10 template <> struct func_unroll<0u> {
11     template <typename F>
12     static inline void call(F const&){}
13 };
14
15
16 struct do_unroll {
17     template <typename F, unsigned unrollDelta>
18     static inline void run(F const& f,
19     int &baseStep, int &restSteps) {
20         for (int j = baseStep; j; --j) {
21             func_unroll<unrollDelta>::call(f);
22         }
23         for (int j = restSteps; j; --j) {
24             f();
25         }
26     }
27 };
```

Fig. 1. Loop unrolling with C++ templates.

Passing appropriate parameters to do_unroll<...>::run(...) from (see Fig. 1), one may call function f(...) baseStep×unrollDelta+restSteps times avoiding unrollDelta−1 counter comparisons and decrementations. Achieved performance improvement is discussed in more details in [15].

```
1 for (int i_fRows = fRows; i_fRows; --i_fRows) {
2     for (int j_fClns = fCols; j_fClns;
3     --j_fClns) {
4         __asm__ __volatile__ (
5             "vdup.32      d0, %[filter]       \n"
6             "vld1.32      {q2,q3}, [%[src]]   \n"
7             "vmovl.u8     q8,  d4             \n"
8             "vmovl.u8     q9,  d5             \n"
9             "vmovl.u8     q10, d6             \n"
10            "vmovl.u8     q11, d7             \n"
11            "vmla.u16     q12, q8 , d0[0]     \n"
12            "vmla.u16     q13, q9 , d0[0]     \n"
13            "vmla.u16     q14, q10, d0[0]     \n"
14            "vmla.u16     q15, q11, d0[0]     \n"
15            :: [filter] "r" (*filter++)
16            , [src]    "r" (src++)
17            : "memory");
18    }
19    src = src - fCols + cols;
20 }
```

Main cycles. Implicit comparison with zero saves 10%-15% processing time.

8 to 16 bit data conversion (data from q2 and q3 is distributed over q8...q11). Multiplying 16 elements from q8...q11 by d0[0] and accumulating result in q12...q15.

Fig. 2. CO optimization with SIMD NEON32

2.3 Optimization by means of special algorithms

Instead of rather general consideration as we did previously, let us focus on CO. The main obstruction for SIMD optimization is translating floating point CO to fixed-point with acceptable loss of precision. First of all, SIMD operations are performed on integers only.

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

$$\Gamma_{i,j} = \nu\gamma_{i,j}, \quad \nu \in \mathbb{R}, \quad \gamma_{i,j} \in \mathbb{Z} \tag{2}$$

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

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

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

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

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

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

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

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

What we propose is selecting for given Γ the biggest ν possible, such that γ still satisfies (3) or (4) (which one depends on whether kernel is purely positive or not). Of coarse, we should be concerned, whether there exist any useful kernels that can be reduced to a suitable form. And it seems there are plenty of them.

In conclusion, modern hardware provides mechanisms for vectorization, i.e. SIMD technologies, that can be used by programmers to enhance performance of the application. Most cases this technology is utilized by compiler to generate binary code without participation of programmer. Suitable choose of 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. Developing assembly code, one should represent the function in suitable for SIMD optimization form. It is not always possible and often restrictions (3,4) should be satisfied. In the next section we will provide new method of CO optimization and then compare it with existing results, e.g. OpenCV and ARM CL.

```
1  __asm__ __volatile__ (
2      "vmovl.u16      q8 , d24                 \n"
3      "vmovl.u16      q9 , d25                 \n"
4      "vmovl.u16      q10, d26                 \n"
5      "vmovl.u16      q11, d27                 \n"
6      "vmovl.u16      q12, d28                 \n"
7      "vmovl.u16      q13, d29                 \n"
8      "vmovl.u16      q14, d30                 \n"
9      "vmovl.u16      q15, d31                 \n"
10     "vcvt.f32.u32   q8 , q8                  \n"
11     "vcvt.f32.u32   q9 , q9                  \n"
12     "vcvt.f32.u32   q10, q10                 \n"
13     "vcvt.f32.u32   q11, q11                 \n"
14     "vcvt.f32.u32   q12, q12                 \n"
15     "vcvt.f32.u32   q13, q13                 \n"
16     "vcvt.f32.u32   q14, q14                 \n"
17     "vcvt.f32.u32   q15, q15                 \n"
18     "vmul.f32       q8 , q8 , d3[0]          \n"
19     "vmul.f32       q9 , q9 , d3[0]          \n"
20     "vmul.f32       q10, q10, d3[0]          \n"
21     "vmul.f32       q11, q11, d3[0]          \n"
22     "vmul.f32       q12, q12, d3[0]          \n"
23     "vmul.f32       q13, q13, d3[0]          \n"
24     "vmul.f32       q14, q14, d3[0]          \n"
25     "vmul.f32       q15, q15, d3[0]          \n"
26     "vcvt.u32.f32   q8 , q8                  \n"
27     "vcvt.u32.f32   q9 , q9                  \n"
28     "vcvt.u32.f32   q10, q10                 \n"
29     "vcvt.u32.f32   q11, q11                 \n"
30     "vcvt.u32.f32   q12, q12                 \n"
31     "vcvt.u32.f32   q13, q13                 \n"
32     "vcvt.u32.f32   q14, q14                 \n"
33     "vcvt.u32.f32   q15, q15                 \n"
34     "vqmovn.u32     d31 , q15                \n"
35     "vqmovn.u32     d30 , q14                \n"
36     "vqmovn.u32     d29 , q13                \n"
37     "vqmovn.u32     d28 , q12                \n"
38     "vqmovn.u32     d27 , q11                \n"
39     "vqmovn.u32     d26 , q10                \n"
40     "vqmovn.u32     d25 , q9                 \n"
41     "vqmovn.u32     d24 , q8                 \n"
42     "vqmovn.u16     d31 , q15                \n"
43     "vqmovn.u16     d30 , q14                \n"
44     "vqmovn.u16     d29 , q13                \n"
45     "vqmovn.u16     d28 , q12                \n"
46     "vst1.32        {q14,q15},[%[dst]]!      \n"
47     :: [dst] "r"(dst) : "memory");
```

16 to 32 bit conversion (16-bit data from q12...q15 registers is converted to unsign[ed] 32-bit and placed into q8...q15) (lines 2–9)

32-bit unsigned integers from q8...q15 ar[e] converted to 32-bit floating-point. (lines 10–17)

Data normalization with coefficient ν (stored in q1, more specifically – d3). (lines 18–25)

Unsigned 32-bit to unsigned 8-bit conversion. (lines 26–45)

← Storing data, final steps for CO (line 46)

Fig. 3. Normalization procedure with SIMD NEON32

3 Optimization of Convolution Operation by means of SIMD

In current contribution we propose a new method of Convolution Operation (CO) optimization based on SIMD technique. We presume target kernel to satisfy condition (3). In this section we will provide all necessary considerations and assembly code that illustrates proposed approach. Next section will be devoted to experimental comparison of this method's performance to known CO implementations (OpenCV and ARM CL).

Regarding condition (4), provided code should be just slightly modified. We will avoid redundant listings and provide code considering only condition (3), while at the end of the section all necessary modification for condition (4) will be described.
We start with basic implementation of CO, (see Fig. 2). It contains no specific optimizations, but still is a good point to start our considerations.

Here q_n are ARM-NEON registers, regarding syntax and instructions order we will strictly follow ARM reference manuals. For the sake of simplicity we avoided normalization by coefficient ν in (see Fig. 2), but for completeness let us provide it as a separate (see Fig. 3).

In (see Fig. 3) we suppose data for normalization to be stored in registers q12 … q15, while d3 contains normalization coefficient ν. Presented code is in some sense multipurpose and may be used with different CO implementations.

Now we switch gears to the CO optimization itself. In (see Fig. 2) we provided some initial version of this operation in assembly code. But it has one significant

drawback - slow data loading. Following (see Fig. 4) avoids this problem by using one of the registers as buffer. It is known, that simultaneous loading of 16 bytes is quicker than loading them one-by-one (approximately 10 and 40 cycles respectively). Thus we use one register for preloading extra data and then use this data byte-by-byte without redundant load operations.

```
1  for (int __iFltR = filterR, __i = 0; __iFltR; --__iFltR, __i += filterC) {
2      __asm__ __volatile__ (
3      "vld1.32        {q0,q1}, [%[src]]     \n"  <-- Loading 32 bytes of grayscale image to q0, q1
4      "vld1.32        {q2}, [%[src1]]       \n"  <-- Loading next 16 bytes of grayscale image to buffer q2
5      "vld1.32        {q3}, [%[flt]]        \n"  <-- Loading 16 bytes of CO kernel in q3
6      ::[flt] "r" (flt + __i)
7      , [src] "r" (src)
8      , [src1] "r" (srcPlus32)
9      : q0_to_q7, q8_to_q15, "memory", "cc");
10     for (register int __jFltC = fltCdiv8; __jFltC; --__jFltC) {
11         __asm__ __volatile__ (
12         "vdup.8         d16,d6[0]             \n"
13         "vmlal.u8       q12, d0, d16          \n"
14         "vmlal.u8       q13, d1, d16          \n"
15         "vmlal.u8       q14, d2, d16          \n"
16         "vext.8         q0,q0,q1,#1           \n"
17         "vmlal.u8       q15, d3, d16          \n"
18         "vext.8         q1,q1,q2,#1           \n"
19         "vdup.8         d17,d6[1]             \n"
20         "vext.8         q2,q2,q3,#1           \n"
21
22         "vmlal.u8       q12, d0, d17          \n"
23         "vmlal.u8       q13, d1, d17          \n"
24         "vmlal.u8       q14, d2, d17          \n"
25         "vext.8         q0,q0,q1,#1           \n"
26         "vmlal.u8       q15, d3, d17          \n"
27         "vext.8         q1,q1,q2,#1           \n"
28         "vdup.8         d16,d6[2]             \n"
29         "vext.8         q2,q2,q3,#1           \n"
30
31         "vmlal.u8       q12, d0, d16          \n"
32         "vmlal.u8       q13, d1, d16          \n"
33         "vext.8         q2,q2,q3,#3           \n"
34         "vmlal.u8       q14, d2, d16          \n"
35         "vext.8         q0,q0,q1,#1           \n"
36         "vmlal.u8       q15, d3, d16          \n"
37         "vext.8         q1,q1,q2,#1           \n"
38         "vext.8         q2,q2,q3,#1           \n"
39         ::: q0_to_q7, q8_to_q15, "memory", "cc");
40     }
41     for (register int __jFltC = fltCmod8; __jFltC; --__jFltC) {
42         __asm__ __volatile__ (
43         "vdup.8         d16,d6[0]             \n"
44         "vmlal.u8       q12, d0, d16          \n"
45         "vmlal.u8       q13, d1, d16          \n"
46         "vmlal.u8       q14, d2, d16          \n"
47         "vext.8         q0,q0,q1,#1           \n"
48         "vmlal.u8       q15, d3, d16          \n"
49         "vext.8         q1,q1,q2,#1           \n"
50         "vext.8         q2,q2,q3,#1           \n"
51         "vext.8         q2,q2,q3,#1           \n"
52         :::q0_to_q7, q8_to_q15, "memory", "cc");
53     }
54     src += srcCols;
55     srcPlus32 += srcCols;
56 }
```

8-bit to 16-bit conversion and multiplication with kernel. (lines 12–38)

Here q0 and q1 registers contain part of the image that should be convolved with kernel stored in q3. Since loading 16 bytes of data one-by-one takes approximately ≈ 40 cycles, while loading 16 bytes at once is worth approximately ≈ 10 cycles, register q2 is exploited as buffer for 16 more bytes of input image to speedup. Data from buffer is being used by cyclically shifting content of q0, q1, and q2 byte-by-byte (performed with *vext* command). Code yields one byte of unnormalized resulting image.

Replicates listing 2 but with data loading operation (lines 43–51)

Fig. 4**.** CO optimization with SIMD ARM-NEON

The main feature of the presented approach (see Fig. 4) is usage of cyclic shift (i.e. vext.8 q0,q0,q1,\#1) that allows kernel buffering and thus we need less "loading operations" (for more details please see comments in (see Fig. 4). Worth noting, that provided (see Fig. 4) demands kernel containing not more than 16 elements in one row. If we need kernels with more than 16 elements in a row, the listing should be just slightly modified.

As we mentioned earlier, this code works for kernels satisfying condition (3). To make it applicable to kernels satisfying (4) we need to change all vmlal.u8/u16 operations to vmlal.s8/s16. This small but crucial changes transform (see Fig. 4) into code capable of working with signed integer kernels. Depending on given kernel, one can choose between this two options.

In conclusion, we found a class of kernels that allow significant optimization of CO by means of SIMD and were able to implement appropriate code combining approaches of loop unrolling and method from [13]. Exploiting significant difference in time for simultaneous 16 byte loading compared to one-by-one loading, we were able

to achieve significant speedup of CO. More detailed results and consideration of 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 one we chose functions cv::filter2d(...) from OpenCV library and NEConvolution{N}x{N}::run() from ACL library. The latter is well-known among AI and DIP researchers due to its high-quality and optimized code.

For comparison we used latest stable tags available at the moment we started research, release tags are 4.0.0 (2018-11-18 [11:08:36]) for OpenCV and v19.02 (2019-02-28 [14:25:18]) for ACL. Compilation was performed with NDK-r18b - latest stable NDK at that moment for to achieve API capability between them. We ensured that libraries utilize vectorization compiling them with flags ANDROID_ABI=armeabi-v7a with NEON ANDROID_NATIVE_API_LEVEL=22 CPU_BASELINE/CPU_BASELINE_FINAL=NEON CPU_BASELINE_FLAGS=-mfpu=neon -O3 -DNDEBUG. Both OpenCV and ACL were linked as static libraries.

Devices. To make our measurements more relevant we used a set of different devices. This helps us to understand the influence of architecture, CPU series, and other parameters on the execution time. Following table lists devices we have used and parameters of their CPUs.

Table 1. Devices that participated in the experiment.

CPU	Architecture	Series*	Device
Exynos 4	armeaby-v7a	Cortex-A9 x 4	Samsung GS III
MT6752	arm64-v8a	Cortex-A53 x 8	Lenovo P70A

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 Android OS. To avoid this problem we used the following procedure: each function (cv::filter2d(...), NEConvolution{N}x{N}::run(), and newCO(...)) was successively called 3 times (for robustness and to simulate RGB processing) and result was stored to array. After collecting 35 data-points we calculated median value and treated it as trice the execution time of the function under consideration.

Kernel sizes varied 2×2, 3×3, … , 15×15 for experiments with our implementation and cv::filter2d(...), while implementation of NEConvolution{N}x{N}::run() necessitates usage of odd-sized kernels only, e.g. 3×3, 5×5, etc. Digital images (DIs) were generated with equal width and height, 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 integer

part of the number. Results are further presented in form of fractions cv::filter2d(...) execution time divided by execution time of our implementation and NEConvolution{N}x{N}::run() execution time divided by execution time of our implementation. Results. First we compared time consumption of the code (see Fig. 4) and reference function cv::filter2d(...), result is presented in figures 5a and 5b.

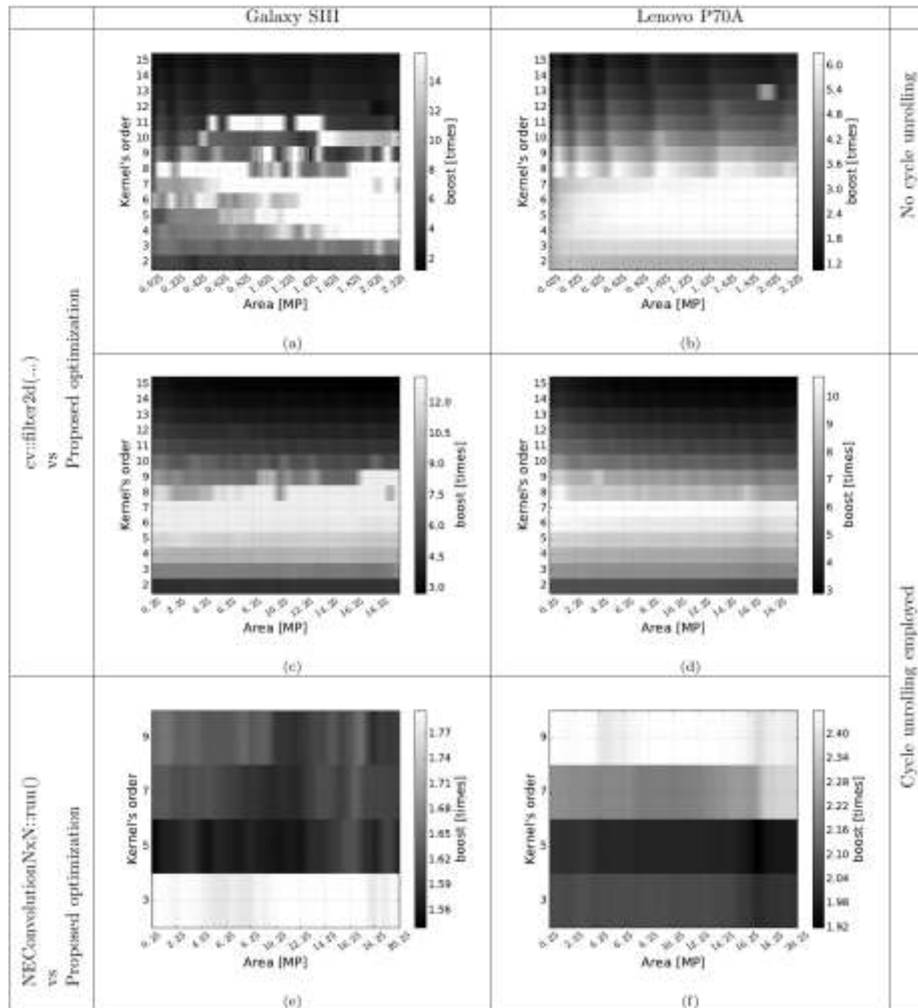

Fig. 5. Performance comparison for different devices and reference functions. Color intensity designates relative time consumption for reference function with regard to proposed method, e.g. acceleration one may achieve by using presented approach instead of the reference function (the brighter is color - 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 proposed method.

As coordinates we use sizes of kernel and image, while color intensity designates acceleration, one may achieve using proposed method instead of the reference method (e.g. fraction of execution times: reference function to proposed).

Despite presented results demonstrate advantage of the proposed method, there is still room for improvement. It seems, compiler is unable to unroll cycles effectively on its own,- one may check this by compiling presented code and exploring binary with any suitable disassembler (e.g. IDA or objdump tool). Thus, we may achieve additional 30%-40% of acceleration by utilizing techniques [12-13].

Results for the modified code are shown in figures 5c-5f. We compared time consumption of the (see Fig. 4), modified with approaches [12-13], and both reference functions (cv::filter2d(...) and NEConvolution{N}x{N}::run()). Besides, we varied image sizes up to 4500×4500 (~20 [MP]) to emulate modern cameras.

As (fig. 5) suggests, acceleration is independent (almost) on 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 1) may be noted for big kernels (9×9 to 15 ×15). Regarding mean acceleration, it is estimated as approximately 1.7 times.

It is worth noting, we did not use parallelism for acceleration. Employing OpenMp or implementing parallelism by any other means may improve presented results twice or even more. Moreover, no preprocessing, e.g. image tiling, was performed. Probably, this technique may increase performance of the approach as well [16-17].

5 Conclusion

In conclusion, we propose a method of convolution operation (CO) acceleration. We show that many kernels utilized for practical applications can be reduced to integer form (table I) that allows for SIMD optimization usage. Despite SIMD itself leads to a significant boost of performance, we were able to push the frontiers even further by exploiting significant difference in time for simultaneous 16 byte loading (approximately 10 cycles) compared to their one-by-one loading (approximately 40 cycles) - q2 register is used as a buffer and loading operations are partially substituted with cyclic shift (see Fig. 4).

To test the approach we performed comparison with cv::filter2D(...) function from OpenCV library and with NEConvolution{N}x{N}::run() from ACL library (fig. 5). Our results suggest, the current approach leads to significant speedup (mean values: ~1.7× compared to OpenCV and ~1.5× compared to ACL). 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 9×9 we were able to achieve ×4.5 acceleration (compared to cv::filter2D(...) function from OpenCV), while for larger kernels presented approach allows only ×1.5 speedup. We did not use parallelism in our code, thus additional ×2 or more acceleration is possible by employing appropriate techniques, e.g. OpenMp library.

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

References

1. Chyrkov, A., Prystavka, P.: Suspicious Object Search in Airborne Camera Video Stream. In: Hu Z. et al. (eds) Advances in Computer Science for Engineering and Education. ICCSEEA 2018. Advances in Intelligent Systems and Computing, vol 754, pp. 340–348. Springer, Cham, Switzerland (2018)
2. S. Gnatyuk, V. Kinzeryavyy, M. Iavich, D. Prysiazhnyi, Kh. Yubuzova, High-Performance Reliable Block Encryption Algorithms Secured against Linear and Differential Cryptanalytic Attacks, CEUR Workshop Proceedings, Vol. 2104, pp. 657-668, 2018.
3. "Documentation for open-cv," https://docs.opencv.org/trunk/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04, 2017, [Online; accessed 27-November-2017].
4. M. J. Flynn, "Very high-speed computing systems," Proceedings of the IEEE, vol. 54, no. 12, pp. 1901–1909, 1966.
5. "Arm® Cortex® - a53 mpcore processor: Reference book of cortex-a53 cpus,"http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf, 2013, [Online; accessed 27-November-2017].
6. "Qualcomm extends hexagon dsp," http://pages.cs.wisc.edu/~danav/pubs/qcom/ hexagon_microreport2013_v5.pdf, 2013,[Online; accessed 27-November-2017].
7. "Qualcomm hexagon dsp: An architecture optimizedfor mobile multimedia and communications," https://developer.qualcomm.com/download/hexagon/hexagon-dsp-architecture.pdf, 2013, [Online; accessed 27-November-2017].
8. Griffith, GCC: the complete reference. McGraw-Hill, Inc., 2002.
9. B. C. Lopes and R. Auler, Getting started with LLVM core libraries. Packt Publishing Ltd, 2014.
10. "Presentation of arm-cl," https://community.arm.com/graphics/b/blog/posts/arm-compute-library-for-computer-vision-and-machine-learning-now-publicly-available, 2017, [Online; accessed 24-May-2017].
11. "Video presentation for arm-cl," https://developer.arm.com/technologies/compute-library?_ga=2.909169.1792656346.1530630636-1257957724.1521634632, 2017, [Online; accessed 24-May-2017].
12. A. Nicolau, "Loop quantization: unwinding for fine-grain parallelism exploitation," Cornell University, Tech. Rep., 1985.
13. J. Xue, "Loop tiling for parallelism, volume 575 of kluwer international series in engineering and computer science," 2000.
14. T. Veldhuizen, "Template metaprograms. c++ report," 1995.
15. V. Sarkar, "Optimized unrolling of nested loops," in Proceedings of the 14th International Conference on Supercomputing, ser. ICS '00. New York, NY, USA: ACM, 2000, pp. 153–166. [Online]. Available: http://doi.acm.org/10.1145/335231.335246
16. Fedushko S., Benova E. Semantic analysis for information and communication threats detection of online service users. The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2019) November 4-7, 2019, Coimbra, Portugal. Procedia Computer Science, Volume 160, 2019, Pages 254-259.
17. Gnatyuk S., Akhmetov B., Kozlovskyi V., Kinzeryavyy V., Aleksander M., Prysiazhnyi D. New Secure Block Cipher for Critical Applications: Design, Implementation, Speed and Security Analysis, Advances in Intelligent Systems and Computing, Vol. 1126, pp. 93-104, 2020.