Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms

Gaurav Mitra, Beau Johnston,
Alistair P. Rendell, and Eric McCreath
Research School of Computer Science
Australian National University
Canberra, Australia
{Gaurav.Mitra, Beau.Johnston, Alistair.Rendell, Eric.McCreath}@anu.edu.au

Jun Zhou
School of Information and
Communication Technology
Griffith University
Nathan, Australia
jun.zhou@griffith.edu.au

***Abstract*—Augmenting a processor with special hardware that is able to apply a Single Instruction to Multiple Data (SIMD) at the same time is a cost effective way of improving processor performance. It also offers a means of improving the ratio of processor performance to power usage due to reduced and more effective data movement and intrinsically lower instruction counts.**

This paper considers and compares the NEON SIMD instruction set used on the ARM Cortex-A series of RISC processors with the SSE2 SIMD instruction set found on Intel platforms within the context of the Open Computer Vision (OpenCV) library. The performance obtained using compiler auto-vectorization is compared with that achieved using hand-tuning across a range of five different benchmarks and ten different hardware platforms. On the ARM platforms the hand-tuned NEON benchmarks were between 1.05× and 13.88× faster than the auto-vectorized code, while for the Intel platforms the hand-tuned SSE benchmarks were between 1.34× and 5.54× faster.

***Keywords*-SIMD; Vectorization; SSE; NEON; AVX; Low-Power; ARM;**

I. Motivation

Arguably the Cray-1 has been the most successful supercomputer to date. The first Cray-1 was installed at Los Alamos National Laboratory in the US in 1976. At the time it held a considerable performance advantage over other machines due to the fact that the hardware was designed and built to perform a Single Instruction on Multiple (independent) Data (SIMD) at the same time.

Since 1976 SIMD "vectorizing" technology has been an integral part of computer development. This is evident today in the existence of Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) as part of the instruction set supported by the ubiquitous Intel x86 Complex Instruction Set Computer (CISC) processor. It is evident in Reduced Instruction Set Computer (RISC) processors such as the SPARC64 VIIIFX [1] and the PowerPC A2 [2] that are used in the Fujitsu K and IBM BlueGene/Q supercomputer systems. SIMD operations are also central to heterogeneous computer systems such as the CellBE processor that was developed by Sony, Toshiba and IBM and that is used in the PlayStation gaming console [3]. These operations are closely related to the Single Instruction Multiple Thread (SIMT) instructions supported on NVIDIA CUDA GPU systems [4].

In 2009 embedded processor designer ARM introduced enhanced floating point capabilities together with NEON SIMD technology into its Cortex-A series of microprocessors [5]. These processors were designed with the smartphone market in mind. Inclusion of improved floating point and SIMD instructions recognized the rapidly increasing computational demands of modern multimedia applications. In this environment SIMD is attractive since it leads to reduced and more effective data movement and a smaller instruction stream with corresponding power savings. The current range of Cortex-A microprocessors are capable of performing single precision floating point operations in a SIMD fashion and are scheduled to support double precision operations in the upcoming 64 bit architecture, ARM v8-A [6].

The move by mobile processor designers to provide better support for computationally intensive floating point operations coincided with high end supercomputers needing to limit their power usage due to capacity, availability and cost constraints. As a consequence there is much interest in using the former as building blocks for the latter. To this end it was recently shown that an Apple iPad 2 with a dual core ARM Cortex-A9 can achieve up to 4 GFLOPS/Watt of double precision performance [7], and more generally, that processors could be classified into a three-tier division according to a GFLOPS/Watt metric. In this classification desktop and server processors were in the least efficient tier one achieving 1 GFLOPS/Watt, AMD and NVIDIA GPU accelerators in tier two achieving 2 GFLOPS/Watt, and ARM RISC processors in tier three achieving 4 GFLOPS/Watt.

In addition to being used as standalone processors, ARM cores are also being used in Multi-Processor Systems-on-chips (MPSoC). These heterogeneous systems may combine an ARM core with a graphics processing unit (GPU). The Tegra systems from NVIDIA is one example [8]. In these systems it is possible to use the GPUs along side the CPU on floating point intensive workloads. In spite of including embedded FPUs and GPUs, these MPSoCs are very energy efficient, consuming minimal Watts at peak throughput.

In light of the above, it is of interest to compare and contrast the performance that can be achieved when using SSE and NEON SIMD instructions to accelerate application

performance on a range of (relatively) low-powered Intel and ARM processors. The aim of this paper is to make such a comparison within the context of the OpenCV image processing library. Four application kernels are considered that have different characteristics. The performance of each is measured and contrasted across 4 Intel and 6 ARM platforms. The results obtained when using the gcc compiler with auto-vectorization on unmodified code are compared with those obtained using hand modified routines that explicitly call SSE or NEON intrinsic functions.

In the following section we describe SIMD vector operations and review related work. Section III summarizes the OpenCV image processing algorithms used as benchmarks in this work, and details the various hardware and software environments used. Section IV and V report the results and provide associated discussion. Section VI contains our conclusions and future work.

II. Background and Related Work

Below we explain the characteristics of SIMD operations, how they are used, compare and contrast the Intel SSE2 versus the ARMv7 NEON SIMD intrinsic functions, and describe related work.

A. Understanding SIMD Operations

SIMD operations act on multiple data elements of the same size and type (in a vector) simultaneously. Compared to sequential or scalar processing, using a SIMD instruction can provide a theoretical speed-up equal to the number of elements that can be operated on by a single SIMD instruction. This speed-up depends on the number of elements of a particular data type that can be packed into a single SIMD register. SIMD operations also reduce the total instruction count and enable more efficient data movement which leads to increased energy efficiency [9]. For example, in the context of mobile multimedia applications, audio data usually exists in 16-bit data types, while graphics and video data exists in 8-bit data types. Processors used in mobile multimedia are primarily 32-bit such as the ARM Cortex-A8 and A9. Operations on 8 and 16-bit data are performed in 32-bit registers, using roughly the same energy requirements as when processing 32-bit data quantities. By packing multiple 8 or 16-bit data types into a single 32-bit register and providing SIMD vector operations, the processor can produce multiple results with minimal increase in the required energy use. Further, as many operations are performed at once, the time for the entire computation to complete is shortened, leading to greater reduction in the energy required to produce each result.

A comparison of using scalar and SIMD vector addition is shown in Figure 1. In this figure the left-hand side shows the scalar addition of four elements of vector A, with four elements of vector B to produce four elements of vector C. In this process each element of A and B is first read from memory before being added together, and then written back to the memory associated with C. Separate load, add and store instructions are issued for each element of the vector, giving rise to a total of 16 instructions for a vector of length four. On the right-hand side the same task is performed using SIMD vector operations. In this approach a single load instruction is used to read four elements of A from memory and similarly for B. A single add instruction sums corresponding elements of A and B, with a final single store instruction used to write four elements of C back to memory. Overall, 16 separate instructions are reduced to four, giving a theoretical overall speed-up of four.

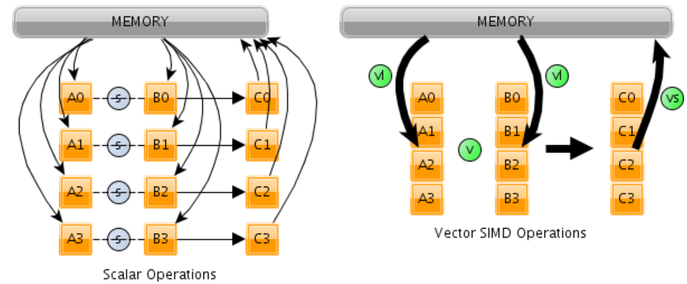

Figure 1: Scalar vs. SIMD Vector Addition

B. Using SIMD Operations

There are three primary ways of utilizing SIMD instructions on Intel and ARM processors: i) writing low-level assembly code; ii) use of compiler supported intrinsic functions; and iii) compiler auto-vectorization. Writing low-level assembly code that use SIMD instructions and available registers is often considered the best approach for achieving high performance. However, this method is time consuming and error prone. Use of compiler supported intrinsic functions provides a higher level of abstraction, with an almost one-to-one mapping to assembly instructions, but without the need to deal with register allocations, instruction scheduling, type checking and call stack maintenance. In this approach intrinsic functions are expanded inline, eliminating function call overhead. It provides the same benefits as inline assembly, improved code readability and less errors [10]. The penalty is that the overall performance boost becomes dependent on the ability of the compiler to optimize across multiple intrinsic function calls. Option (iii) leaves everything to the compiler, relying on it to locate and automatically vectorize suitable loop structures. Therefore, the quality of the compiler and the ability of the programmer to write code which aids auto-vectorization becomes essential. Unfortunately, a recent study [11] showed that state-of-the-art compilers were able to vectorize only 45-71% of synthetic benchmarks and 18-30% of real application codes. In particular it was noted that compilers did not perform some critical code transformations necessary for facilitating auto-vectorization. Non-unit stride memory access, data alignment and data dependency transformations were found to be pertinent issues with compiler auto-vectorization. Our work analyzes similar issues but concentrates on using intrinsic functions to perform SIMD operations, and provides benchmark speed-ups compared against those achieved using auto-vectorizing compiler options on the original code.

C. NEON and SSE Operations

Below we briefly outline details of the processor registers, intrinsic data types, and intrinsic functions that are supported by the NEON and SSE2 instruction sets.

1) Processor Registers: Prior to SIMD operation execution, data elements of the same size and type must be fetched from memory into specific processor registers. As shown in Figure 1, elements A0 to A3 and B0 to B3 must fit into the special SIMD registers before the vector `v` operation can be performed. Vector load operation, `vl` and store operation, `vs` are used to populate the registers. The ARMv7 architecture [12] defines an extension register set apart from core registers to cater to Advanced SIMD (NEON) and VFPv3 floating point operations. There are thirty-two 64-bit double-word registers, D0-D31, usable by NEON and VFPv3 operations. These registers can also be viewed by NEON operations as sixteen 128-bit quad-word registers, Q0-Q15. Intel SSE2 operations work with 128-bit wide XMM registers and 64-bit wide MMX registers. There are eight XMM registers, XMM0-XMM7 and eight MMX registers, MM0-MM7. The newer AVX and AVX2 instruction use 256-bit wide YMM registers.

2) Intrinsic Data Types: Separate data types are defined to represent vectors used as operands in intrinsic functions. These types can only be used as parameters, assignments or return values for intrinsic functions and cannot be used in arithmetic expressions. In C code, Intel SSE2 uses four types to represent packed data elements [10]: i) `__m64` can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value; ii) `__m128` can hold four packed single precision 32-bit floating point values or a scalar single precision floating point value; iii) `__m128d` can hold two double precision 64-bit floating point values or a scalar double precision floating point value; and iv) `__m128i` can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values. ARMv7 NEON data types [5] are more descriptive and are named according to the pattern: $< type >< size > x < number\ of\ lanes > _t$. Types representing integer data include `int8x8_t`, `int8x16_t`, `int16x4_t`, `int16x8_t`, `int32x2_t`, `int32x4_t`, `int64x1_t` and `int64x2_t`. Unsigned integer data is represented by similar types with `uint` instead of `int` as the $< type >$. There are 2 floating point types, `float32x2_t` and `float32x4_t`, both for single precision floating point values. NEON also provides 8 and 16-bit polynomial data types, `poly8x8_t`, `poly8x16_t`, `poly16x4_t` and `poly16x8_t`. Further, NEON also provides data types representing arrays of vector types following the pattern: $< type >< size > x < number\ of\ lanes > x < length\ of\ array > _t$. These types are treated as ordinary C structures. For example, `int16x4x2_t` represents a struct type with parameter `int16x4_t val[2]`. Here, `val[2]` is an array of size 2 holding `int16x4_t` data.

3) Naming and classification of intrinsic functions: All intrinsic functions follow a naming convention. SSE2 intrinsics are named according to the pattern: $_mm_[intrin_op]_[suffix]$. Here, *intrin_op* refers to the basic operation of the intrinsic such as `add` or `sub`, while *suffix* denotes the type of data on which the intrinsic operates [10]. NEON intrinsics follow: $[intrin_op][flags]_[type]$. The flag `q` denotes that the intrinsic is using quad-word (128-bit) Q registers [5]. SIMD intrinsic functions can be classified under eight categories:

a) *Data Movement*: Loads and stores of consecutive data elements of same type and size between SIMD registers and memory. Both SSE2 and NEON support many different load/store combinations for single vectors of each supported data type. NEON further supports load/stores between arrays of vectors.

b) *Arithmetic*: Addition, subtraction, multiplication, absolute difference, maximum, minimum, and saturation arithmetic are provided by both SSE2 and NEON. Reciprocal and square root operations are also provided by both for floating point data. SSE2 also supports division and averaging instructions for double precision floating point data. NEON further implements different combinations of these operations such as halving add; multiply accumulate; multiply subtract; halving subtract; absolute difference and accumulate; folding maximum, minimum; operations with a scalar value; and single operand arithmetic.

c) *Logical*: Bitwise AND, OR, NOT operations and their combinations. Both SSE2 and NEON provide AND, OR and XOR operations. Further, SSE2 provides AND NOT operations and NEON provides NOT, bit clear, OR complement and bitwise select operations.

d) *Compare*: Data value comparisons $=, <=, <, >=, >$. Both NEON and SSE2 provide these operations. Further, SSE2 provides the $\neq$ (neq) operation and NEON provides absolute value $>, >=, <, <=$ operations.

e) *Pack, Unpack and Shuffle*: SSE2 and NEON both provide *set* instructions that store data elements into vectors. NEON also supports setting individual items or lanes in vectors, and initializing vectors from bit patterns. SSE2 supports *unpack* and *pack* operations that interleave high or low data elements from input vectors and place them in output vectors. NEON has similar vector extract operations. SSE2 also provides *shuffle* operations that rearrange bits based on a control mask.

f) *Conversion*: Transformations between supported data types. Both SSE2 and NEON provide conversions between floating point and integer data types with saturation arithmetic and low/high element selection within vectors.

g) *Bit Shift*: Often used to perform integer arithmetic operations such as division and multiplication. SSE2 and NEON both have bit shifts by a constant value. NEON further has bit shifts by signed variables; saturation and rounding shifts; and shifts with insert.

h) *Miscellaneous*: Cache specific operations and other applications. SSE2 has casting, bit insert, cache line flush, data prefetch, execution pause and others. NEON has endianness swap, vector reinterpret cast, transpose operations, table lookups and others.

D. Related Work

Several works address performance improvement of audio/video signal processing using NEON compiler intrinsics on ARM platforms, and SSE or AVX intrinsics on Intel platforms. Infinite Impulse Response (IIR) filters, commonly used as low/high pass filters in audio/video processing, have been accelerated up to 2× using NEON in [13] and from 1.5× to 4.5× using SSE in [14]. Video encoding/decoding is improved in multiple works using NEON. These include AVS decoding up to 2.11× in [15], H.264 decoding up to 1.6× in [16], [17], and MPEG-4 encoding and decoding up to 2× in [16], [17]. The Fast Fourier Transform (FFT), a core algorithm for many audio/video and image processing applications was optimized to used SSE, AVX and NEON instructions to create an alternative implementation, SFFT in [18]. It was benchmarked against popular FFT implementations such as FFTW and SPIRAL and was shown to be at least as fast or faster than the state-of-the-art on Intel and ARM platforms. In an evaluation of AVX instructions for high performance computing, [19] demonstrated that AVX delivers between 1.58× and 1.88× improvement over SSE 4.2 in computationally intensive benchmarks such as LINPACK and HPCC. Significant performance advantages of using inline assembly to program SSE 4.2 and AVX instructions compared to compiler auto-vectorized code were observed in [20]. Data mining applications were used to compare single precision and double precision floating point performance improvements from SSE and AVX instructions in [21] and [22]. It was shown that AVX intrinsics performed at least 1.63× better than SSE for single precision computations and at least 1.71× better for double precision. Several computer vision routines were optimized using NEON and benchmarked on an NVIDIA Tegra 3 device in [23]. The objective of this work was to measure comparative improvements offered by NEON versus those offered by OpenGL shading language using the on-chip GPU on the Tegra 3 processor. Impressive results were demonstrated including speed-ups of 23× for median blur, 4.6× for Gaussian blur, 9.5× for color conversion, 3.1× for sobel filtering, 1.6× for canny edge detection, and 7.6× for image resizing. Apart from SFFT [18], no other work was found comparing performance improvements of SIMD intrinsic functions across Intel and ARM platforms.

III. Experimental Evaluation

To investigate the ability of NEON operations in providing performance speed-ups with comparison to SSE2 operations, we use a set of image processing benchmarks from the popular OpenCV library [24]. It contains over 500 functions for computer vision related applications, capable of performing various image processing tasks on most common image formats. We hand optimize certain functions (with existing SSE2 optimizations) in two fundamental OpenCV modules, *Core* and *Imgproc* using NEON intrinsic functions. These functions then form the basis of our separate image processing benchmarks. In the following subsections we briefly describe these benchmarks, the ARM and Intel platforms evaluated, the compilers and tools utilized, how the experiments are conducted and the test images used. A small code snippet is provided to briefly compare between the use of NEON and SSE2 intrinsic functions.

A. Image Processing Benchmarks

1) Conversion of Float to Short: A fundamental image processing operation is that of converting image pixel values from integer to floating point formats, for further filtering or transformations. After desired operations, these values are then converted back to integers for the processed images to be displayed. In many cases when performing floating point to integer conversion there is an overflow, and saturation arithmetic is required. OpenCV provides the template cast operation, *saturate_cast*, to perform this task. This routine is shown below:

```
template<> inline short saturate_cast<short>(float v)
{ int iv = cvRound(v); return saturate_cast<short>(iv); }
template<> inline short saturate_cast<short>(int v)
{
   return (short)((unsigned)(v - SHRT_MIN) <= (unsigned)USHRT_MAX ?
   v : v > 0 ? SHRT_MAX : SHRT_MIN);
}
CV_INLINE int cvRound( double value )
{
    #if (defined _MSC_VER && defined _M_X64) || (defined __GNUC__ && defined __x86_64__ && defined __SSE2__ && !defined __APPLE__)
    __m128d t = _mm_set_sd( value );
    return _mm_cvtsd_si32(t);
    #else
    return (int)(value + (value >= 0 ? 0.5 : -0.5));
    #endif
}
```

We hand optimized the float to short saturate cast operation using NEON intrinsics. The critical loop of this operation, the SSE2 and our NEON optimized versions are shown below:

```
/* OpenCV un-optimized */
for( ; x < size.width; x++ )
    dst[x] = saturate_cast<short>(src[x]);

/* SSE2 */
for( ; x <= size.width - 8; x += 8 )
{
    __m128 src128 = _mm_loadu_ps (src + x);
    __m128i src_int128 = _mm_cvtps_epi32 (src128);

    src128 = _mm_loadu_ps (src + x + 4);
    __m128i src1_int128 = _mm_cvtps_epi32 (src128);

    src1_int128 = _mm_packs_epi32(src_int128, src1_int128);

    _mm_storeu_si128((__m128i*)(dst + x), src1_int128);
}
```

```
    /* NEON */
    for( ; x <= size.width - 8; x += 8 )
    {
        float32x4_t src128 = vld1q_f32((const
            float32_t*)(src + x));
        int32x4_t src_int128 = vcvtq_s32_f32(
            src128);
        int16x4_t src0_int64 = vqmovn_s32(
            src_int128);

        src128 = vld1q_f32((const float32_t*)(src
            + x + 4));
        src_int128 = vcvtq_s32_f32(src128);
        int16x4_t src1_int64 = vqmovn_s32(
            src_int128);

        int16x8_t res_int128 = vcombine_s16(
            src0_int64, src1_int64);
        vst1q_s16((int16_t*) dst + x, res_int128);
    }
```

Consider first the SSE2 code where the image pixels are initially stored in memory as 32-bit floats. These pixels are processed in groups of eight as indicated by the outer `for` loop. The first SSE intrinsic, `mm_loadu_ps(src + x)`, loads four single precision floats into a single 128-bit SSE register. It then converts these to four signed 32-bit integers using the SSE intrinsic, `mm_cvtps_epi32(src128)`. The process is then repeated for the next four image pixels. At this point there are two 128-bit SSE registers each containing four 32-bit signed integers. The next SSE instruction, `_mm_packs_epi32(src_int128, src1_int128)`, takes both 128-bit SSE registers, downcasts all 32-bit integers to 16-bit, and packs the eight resulting values into a single 128-bit SSE register. The final SSE intrinsic, `mm_storeu_si128((__m128i)(dst + x), src1_int128 )`, copies the result back to main memory. The NEON code is similar except for the downcast, which is done in two stages. The first stage processes each 128-bit register separately, casting each of the 32-bit integers to 16-bits and returning a 64-bit result (`int16x4_t src1_int64 = vqmovn_s32(src_int128)`). The second stage, `int16x8_t res_int128 = vcombine_s16(src0_int64,src1_int64)`, involves packing the two 64-bit results into a single 128-bit register prior to copying the result back to main memory. Therefore the NEON code requires two extra intrinsic function calls compared to SSE code.

2) Binary Image Threshold: Binary thresholding is a common image processing operation, which has been widely used for image segmentation [25]. It requires element-wise comparison between a pixel of interest and a predetermined threshold value. Given an array of pixels and a threshold value, an operation occurs on every pixel depending on whether it is above or below the threshold. This is shown in Algorithm 1.

3) Gaussian Blur: Image blurring is an operation that convolves an image with a blurring or smoothing filter. The convolution operation applies the filter across the image. At each image pixel, the filtered outcome is the weighted sum of itself and the neighborhood pixels, where the weights come

Algorithm 1 Pseudocode: Binary Image Threshold

```
for all pixels in Image do
    if pixel ≤ threshold then
        pixel ← threshold
    else
        pixel ← pixel
    end if
end for
```

from the entry of the filter. We use an anisotropic Gaussian filter with standard deviation set to 1.

4) Sobel Filter: This employs the same form of spatial convolution as Gaussian Blur. The difference between the two operations is that instead of convolving with a Gaussian filter, two separable 1-D Sobel filters are used.

Pseudocode applicable to both the Gaussian and Sobel filters is given in Algorithm 2.

Algorithm 2 Pseudocode: Convolution Filtering

for all pixels I in Image **do**
  for all x pixels in width of filter S **do**
    for all y pixels in height of filter S **do**
      centre pixel $I_{(*,*)}$ += $I_{(x,y)} \times S_{(x,y)}$
    end for
  end for
end for

5) Edge Detection: The Sobel Filter is often used for edge detection. After applying a 2-D Sobel filter and a binary thresholding operation, pixels with low gradient intensity are removed.

B. Evaluated Platforms

Four different Intel platforms supporting SSE2 instructions were considered. The Atom D510 was chosen because it is targeted towards the mobile embedded market [26]. The Core 2 Quad Q9400 platform was chosen as a popular representative of Intel's desktop processors. The Core i7 and Core i5, although having different microarchitectures, represent Intel's recent generations of laptop processors which support the AVX instruction set. Our choice of ARM platforms was determined by their support for NEON instructions and the ability for them to be programmed using readily available tools. Six ARM platforms were used, all with NEON. The first three of these were Android smart-phones: i) Samsung Exynos 3110 processor containing an ARM Cortex-A8, part of the Samsung Nexus S; ii) Texas Instruments OMAP 4460, part of the Samsung Galaxy Nexus using a dual core ARM Cortex-A9 processor; and iii) Exynos 4412 quad core Cortex-A9 processor, part of the Samsung Galaxy S3. The next three ran different Linux distributions: iv) Texas Instruments DaVinci processor [27] containing an ARM Cortex-A8, running Angstrom Linux; v) Exynos 4412 quad core Cortex-A9 processor, part of the ODROID-X development platform [28] running Linaro-Ubuntu Linux;

PROCESSOR	CODENAME	Launched	Threads/Cores/Ghz	Cache L1/L2/L3 (KB)	Memory	SIMD Extensions
			INTEL			
Intel Atom D510	Pineview	Q1'10	4/2/1.66	32(I),24(D)/1024/ No L3	4GB DDR2	SSE2/SSE3
Intel Core 2 Quad Q9400	YorkField	Q3'08	4/4/2.66	32(I,D)/3072/ No L3	8GB DDR3	SSE*
Intel Core i7 2820QM	Sandy Bridge	Q1'11	8/4/2.3	32(I,D)/256/8192	8GB DDR3	SSE*/AVX
Intel Core i5 3360M	Ivy Bridge	Q2'12	4/2/2.8	32(I,D)/256/3072	16GB DDR3	SSE*/AVX
			ARM			
TI DM 3730	DaVinci	Q2'10	1/1.ARM Cortex-A8/0.8	32(I,D)/256/ No L3	512MB DDR	VFPv3/NEON
Samsung Exynos 3110	Exynos 3 Single	Q1'11	1/1.ARM Cortex-A8/1.0	32(I,D)/512/ No L3	512MB LPDDR	VFPv3/NEON
TI OMAP 4460	Omap	Q1'11	2/2.ARM Cortex-A9/1.2	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
Samsung Exynos 4412	Exynos 4 Quad	Q1'12	4/4.ARM Cortex-A9/1.4	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
Samsung Exynos 4412	ODROID-X	Q2'12	4/4.ARM Cortex-A9/1.3	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
NVIDIA Tegra T30	Tegra 3, Kal-El	Q1'11	4/4.ARM Cortex-A9/1.3	32(I,D)/1024/ No L3	2GB DDR3L	VFPv3/NEON

Table I: Platforms Used in Benchmarks

and vi) CARMA: CUDA on ARM Development Kit with a quad-core NVIDIA Tegra 3 system-on-chip composed of Cortex-A9 processors running Ubuntu Linux. The key characteristics of all ten platforms are shown in Table I. In this table the entries in the cache column should be interpreted as L1 Cache size (Instruction and Data) /L2 Cache size /L3 Cache size. To counter effects of dynamic frequency scaling, *cpufreq-set* was used to assign the *performance* scaling governor for platforms that supported this feature. We under-clocked the ODROID-X at 1.3Ghz to enable a direct comparison with the Tegra 3, which was also clocked at 1.3Ghz.

C. Code, Compilers and Tools

Our OpenCV test harness for all Intel platforms and three ARM platforms (TI DM 3730, Tegra T30, ODROID-X) was written in C++. For the smart-phones we used OpenCV4Android [29] and the Android NDK [30] to write the test harness. OpenCV 2.4.2 was used, compiled on all platforms using the CMake cross compilation toolchain for single thread execution. The NEON specific cmake toolchains provided from *opencv.org*, were used to compile OpenCV4Android. All NEON optimizations via intrinsic functions were made directly within OpenCV *core* and *imgproc* modules in an analogous fashion to their SSE2 counterparts. These optimizations were turned ON and OFF using the OpenCV function *cv::setUseOptimized(bool onoff)* with the benchmarks labelled accordingly.

GCC 4.6.3 was used to compile both OpenCV and the test harness across all Intel and ARM platforms. The Android SDK level 15 and NDK r8b compiler, using GCC 4.6.x, was used for the Android phones. Use of the gcc compiler essentially treats all Intel platforms equivalently, and likewise for all ARM platforms (there are slight differences between ARM Linux and Android). To trigger auto-vectorization on the Intel platforms, *-O3 -msse -msse2* compiler options were used in all cases. ARM auto-vectorization flags included combinations of the following: *-mfpu=neon -ftree-vectorize -mtune=cortex-a8 -mtune=cortex-a9 -mfloat-abi=softfp -mfloat-abi=hard -O3*. All code was compiled with debugging symbols enabled via the *-ggdb3* compiler flag. To analyze assembly instructions, we used the *objdump* tool to extract code annotated assembly from compiled *.o* object files.

D. Experimental configuration

The image resolutions chosen for our study are commonly used by modern mobile digital cameras. The smallest resolution is 640×480 or 0.3 Mega-pixels (mpx). Other image sizes were 1280×960 (1mpx), 2560×1920 (5mpx) and 3264×2448 (8mpx). Uncompressed bitmap images with corresponding sizes of 1.2MB (0.3mpx); 4.7MB (1mpx); 19MB (5mpx); 23MB (8mpx) were used for all experiments. We cycled through 5 different images of each resolution 25 times, to obtain an average runtime over 100 runs of a benchmark. We chose to traverse 5 different images in succession to minimize caching effects. In the following tables *AUTO* implies compiler auto-vectorized code and *HAND* implies use of hand optimized intrinsic functions. Time is measured in seconds and speed-up reflects how much faster the HAND experiments were compared to the corresponding AUTO cases. A high resolution timer with an an accuracy greater than 10^{-6} seconds was used in all cases.

IV. Results & Observations

A. Benchmark 1: Float to Short Conversion

Results for the float to short integer benchmark are given in Table II. For the smallest image resolution 640×480, the Core i5 has the best absolute time with both SIMD intrinsic HAND and AUTO. For Intel processors the speed-up obtained with HAND varies from 5.27 for the Atom to just 1.34 for the Core 2 Quad. For the ARM processors the speed-up variation is greater, ranging from 13.88 on the Exynos 3110 smart-phone to 3.42 on the Tegra T30 system. As the image size increases the number of operations performed scales with the number of pixels. This is reflected in absolute execution times which in most cases scale almost linearly with image size, with deviations from this behaviour attributed to cache effects.

The SIMD speed-ups achieved across all platforms and for all image sizes are plotted in Figure 2. Within a given processor type the results are remarkably similar for all image sizes. As this benchmarks involves processing four floating point values in one 128-bit wide register one might expect the maximum speed-up to be a factor of four. On several platforms values greater than four are consistently observed. The speed-up metric examines the performance of SSE intrinsic code against machine code generated from

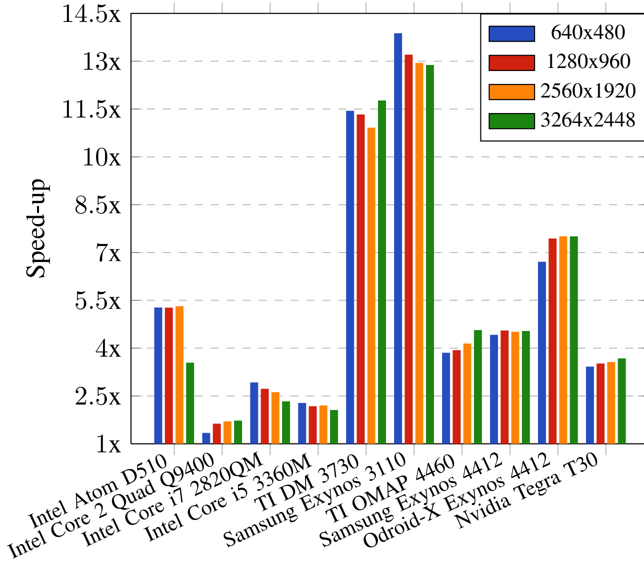

Figure 2: Convert Float to Short relative speed-up factor

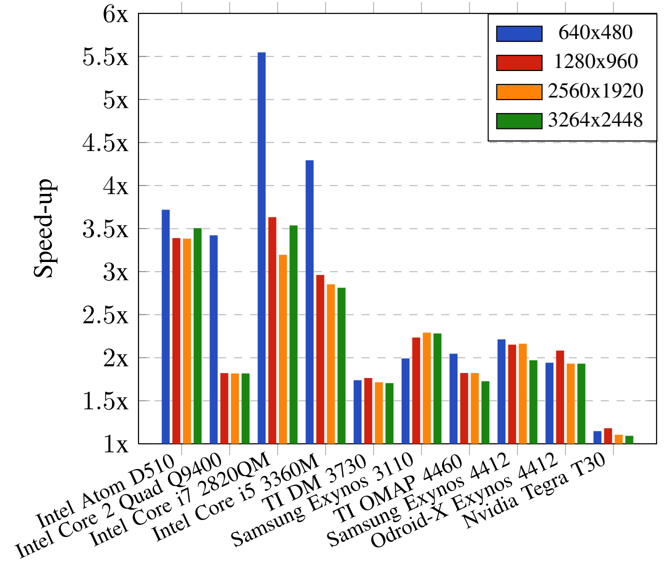

Figure 3: Binary Image Thresholding relative speed-up

the original source. This varies significantly, which explains some of the theoretical speed-ups measured. For example, the SSE intrinsic code stores eight short integers in one aligned store instruction while the C code will perform eight non-aligned memory store operations.

Considering the ARM Cortex-A9 based SoCs, we find as expected, the 1.4Ghz Exynos 4412 in the Galaxy S3 Android smart-phone has the best absolute times on average. For the 1.3Ghz clocked ODROID-X and Tegra T30 A9 systems, although the AUTO times are quite similar, the ODROID outperforms the Tegra for the HAND cases. The ODROID shows more than twice as much benefit from using NEON compared to the Tegra T30 as shown in Figure 2. Comparing ARM SoCs running Android versus those running Linux distributions, we observe the AUTO absolute times for Android platforms to be significantly better on average than those on Linux platforms. This could be attributed to Google's customized gcc 4.6.x and lightweight BIONIC libc libraries used in the Android NDK. HAND times across all ARM platforms except the Tegra T30, maintain consistency when normalized with respect to clock speed.

B. Benchmarks 2-5

Absolute execution times and SIMD speed-ups for the large 3264x2448 image with the Binary Image Thresholding, Gaussian Blur, Sobel Filter and Edge Detection benchmarks are given in Table III. Speed-up results for all image sizes and platforms are presented in Figures 3-6.

The absolute execution times given in Table III increase progressively from benchmark to benchmark reflecting the steadily increasing complexity. Based on processor clock speed the i5 might be expected to give the shortest execution times, and this is indeed the case. However, other architectural differences mean that the 1.66GHz Atom D510 is about 10× slower than the 2.3GHz Intel Core i7. Specifically, in this case the Intel Atom is an in-order processor while the Intel i7 is an out-of-order processor. This enables the i7 processor to execute non-dependent instructions later in the instruction stream before earlier instructions are complete, while the Atom will stall. The fastest ARM system is the 1.4GHz Exynos 4412. This system is typically 8-15× slower than the Intel Core i5. A fairer comparison is, however, between the Exynos 3110 and the Intel Atom D510 as both these systems are in-order processors. In this case we find the Intel to be 3-10× faster than the ARM system. The ODROID consistently outperforms the Tegra T30 in HAND cases, raising questions about what bottlenecks are preventing NEON from performing as well on the Tegra system having the same clock speed and faster DDR3L RAM.

Compared to the 13× SIMD intrinsic speed-up obtained with the float to short benchmark, the maximum speed-up observed in Figures 3-6 is about 5.5 across all platforms. These figures show that in general the benefit of using hand coded SIMD intrinsics over auto-vectorization appears to be slightly greater on the Intel platforms compared with the ARM platforms. The variability of the speed-up also appears to be slightly greater within the Intel platforms compared to within the ARM platforms, even though the same compiler was used in all cases.

V. Discussion

To determine why auto-vectorization does not provide similar benefits to direct use of intrinsic functions, we analyzed the assembly instructions for benchmark 1. We present here only an analysis for the ARM platform. Below is the critical section for the intrinsic optimized NEON assembly followed by the equivalent auto-vectorized assembly.

Image Size	SIMD Intrinsic Optimized	INTEL (SSE2) Atom D510	Core 2 Q9400	Core i7 2820QM	Core i5 3360M	ARM (NEON) TI DM 3730	Exynos 3110	TI OMAP 4460	Exynos 4412	Odroid-X Ex-4412	Tegra T30
640x480	AUTO	0.01492	0.00182	0.00122	0.00090	0.20119	0.13215	0.03145	0.02724	0.04664	0.04865
	HAND	0.00283	0.00136	0.00042	0.00040	0.01758	0.00952	0.00816	0.00616	0.00695	0.01422
	Speed-up	5.27	1.34	2.93	2.28	11.44	13.88	3.86	4.42	6.71	3.42
1280x960	AUTO	0.05952	0.00711	0.00483	0.00358	0.80300	0.49577	0.11285	0.10688	0.18361	0.19347
	HAND	0.01129	0.00436	0.00177	0.00164	0.07087	0.03754	0.02866	0.02347	0.02468	0.05499
	Speed-up	5.27	1.63	2.73	2.18	11.33	13.21	3.94	4.55	7.44	3.52
2560x1920	AUTO	0.23770	0.02845	0.01813	0.01417	3.21380	2.01111	0.44328	0.47170	0.73358	0.80143
	HAND	0.04472	0.01670	0.00692	0.00643	0.29443	0.15534	0.10692	0.10447	0.09770	0.22479
	Speed-up	5.32	1.70	2.62	2.20	10.92	12.95	4.15	4.51	7.51	3.57
3264x2448	AUTO	0.43863	0.06392	0.04412	0.03249	5.28033	3.27790	0.92932	0.75601	1.19228	1.31077
	HAND	0.12374	0.03702	0.01892	0.01578	0.44870	0.25445	0.20347	0.16658	0.15880	0.35630
	Speed-up	3.54	1.73	2.33	2.06	11.77	12.88	4.57	4.54	7.51	3.68

Table II: Time (in seconds) to perform conversion of Float to Short Int

Benchmark	SIMD Intrinsic Optimized	INTEL (SSE2) Atom D510	Core 2 Q9400	Core i7 2820QM	Core i5 3360M	ARM (NEON) TI DM 3730	Exynos 3110	TI OMAP 4460	Exynos 4412	Odroid-X Ex-4412	Tegra T30
BinThr	AUTO	0.06688	0.02010	0.01538	0.01075	0.24731	0.17236	0.12912	0.10356	0.11925	0.13069
	HAND	0.01908	0.01106	0.00435	0.00382	0.14508	0.07553	0.07477	0.05255	0.06176	0.11962
	Speed-up	3.51	1.82	3.54	2.81	1.70	2.28	1.73	1.97	1.93	1.09
GauBlu	AUTO	0.46885	0.12089	0.08392	0.06027	1.28726	0.85978	0.78505	0.50677	0.54349	0.65445
	HAND	0.20303	0.04066	0.02535	0.02034	0.90927	0.54455	0.63407	0.43248	0.50881	0.56470
	Speed-up	2.31	2.97	3.31	2.96	1.42	1.58	1.24	1.17	1.07	1.16
SobFil	AUTO	0.60975	0.16437	0.10417	0.06873	2.06342	1.18199	1.16707	0.82602	0.82258	1.06827
	HAND	0.28214	0.08819	0.05028	0.03653	1.31097	0.57174	0.96051	0.54293	0.54897	0.76793
	Speed-up	2.16	1.86	2.07	1.88	1.57	2.07	1.22	1.52	1.50	1.39
EdgDet	AUTO	0.73730	0.21568	0.15100	0.09142	2.39850	1.36388	1.42742	0.93302	1.00915	1.47294
	HAND	0.40933	0.13984	0.08956	0.06067	1.72260	0.74123	0.92232	0.65695	1.00915	1.18791
	Speed-up	1.80	1.54	1.69	1.51	1.39	1.84	1.55	1.42	1.37	1.24

Table III: Time (in seconds) to perform Binary Thresholding, Gaussian Blur, Sobel Filter and Edge Detection benchmarks on 8mpx (3264x2448) images

```
    /* Intrinsic Optimized ARM Assembly*/
48:   mov r2, r1
    add.w r0, r9, r3   #x+8
    adds  r3, #16     #src+x
    adds  r1, #32     #src+x+4

    //float32x4_t src128= vld1q_f32((const
        float32_t*)(src + x))
    vld1.32 {d16-d17}, [r2]!
    cmp r3, fp
    //int32x4_t src_int128= vcvtq_s32_f32(src128)
    vcvt.s32.f32  q8, q8
    //src128 = vld1q_f32((const float32_t*)(src +
        x + 4))
    vld1.32 {d18-d19}, [r2]
    //src_int128 = vcvtq_s32_f32(src128)
    vcvt.s32.f32  q9, q9
    //int16x4_t src0_int64= vqmovn_s32(src_int128)
    vqmovn.s32  d16, q8
    //int16x4_t src1_int64= vqmovn_s32(src_int128)
    vqmovn.s32  d18, q9
    //int16x8_t res_int128= vcombine_s16(
        src0_int64,src1_int64)
    vorr  d17, d18, d18
    //vst1q_s16((int16_t*) dst + x, res_int128)
    vst1.16 {d16-d17}, [r0]

    //iterate through width if (x <= size.width -
        8)
    bne.n 48 <cv::cvt32f16s(float const*, unsigned
         int, unsigned char const*, unsigned int,
        short*, unsigned int, cv::Size_<int>,
        double*)+0x48>
    //end of if (cv::useOptimized())
```

```
    /* Auto-vectorized ARM Assembly */
    // vector load multiple registers
8e: vldmia     r6!, {s15}
      // vector convert float 64 bit to float 32 bit
      vcvt.f64.f32   d16, s15
      // copy shortened float back to register r0
      vmov   r0, r1, d16
      bl   0 <lrint>
    //perform conversion from float -> int -> short
        using OpenCV's rounding cast:
    //#define  CV_CAST_16S(t) (short)(!(((t)+32768)
        & ~65535) ? (t) : (t) > 0 ? 32767 : -32768)
    add.w r2, r0, #32768   ; 0x8000
    uxth   r3, r0
    cmp    r2, r8
    bls.n b2 <cv::cvt32f16s(float const*, unsigned
        int, unsigned char const*, unsigned int,
        short*, unsigned int, cv::Size_<int>, double
        *)+0xb2>

    cmp r0, #0
    ite gt

    movgt r3, sl
    movle.w r3, #32768   ; 0x8000
b2: adds  r4, #1
    strh.w  r3, [r5], #2
    cmp r4, r7
    bne.n 8e <cv::cvt32f16s(float const*, unsigned
        int, unsigned char const*, unsigned int,
        short*, unsigned int, cv::Size_<int>, double
        *)+0x8e>
```

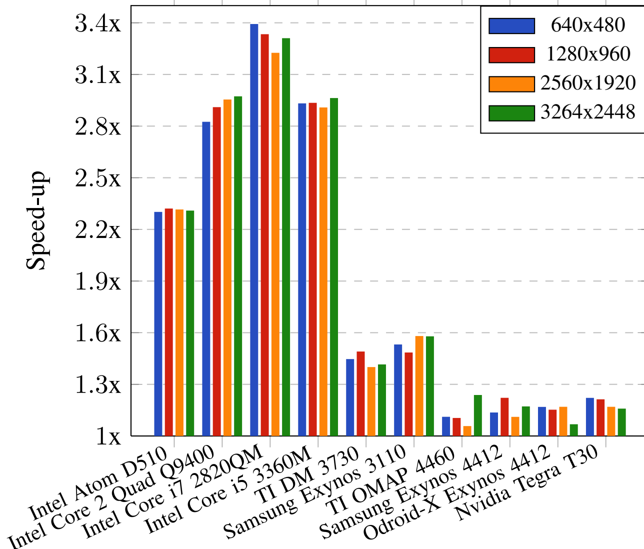

Figure 4: Gaussian Blur relative speed-up factor

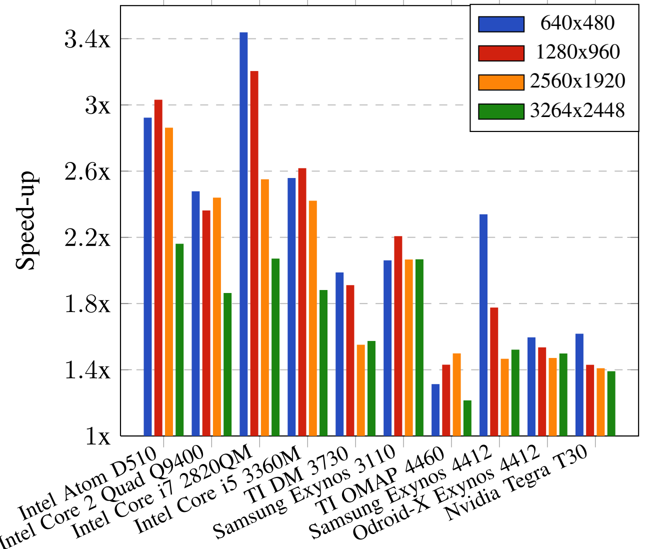

Figure 5: Sobel Filter relative speed-up factor

Mapping the assembly in the first block of code to the NEON intrinsic instructions from which they are generated is fairly straightforward. The one interesting observation is that the `vcombine_s16` has been been replaced with a `vorr` operation, which is a bitwise OR activity. This is unexpected as [12] states that this operation maps to a `vmov` operation. Overall eight NEON intrinsics translate into eight NEON assembly instructions. An additional six other instructions are required to maintain address offsets and control the loop. Thus a total of 14 operations are required per eight output pixels. For the auto-vectorized assembly some use of NEON instructions is apparent, but clearly the major issue is that the loop is not running in blocks of eight pixels. As a consequence many more operations are required per output pixel, explaining the large speed-up that was observed on some of the ARM platforms. Analysis of the SSE code for Intel platforms gives similar findings.

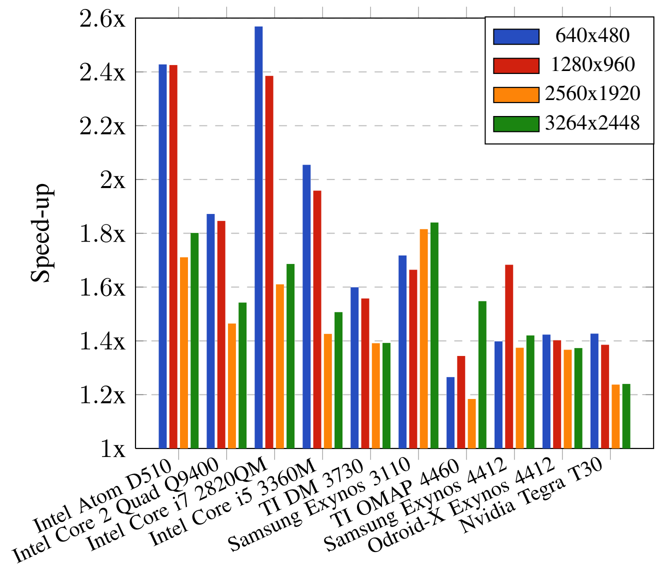

Figure 6: Edge Detection relative speed-up factor

VI. Conclusions and Future Work

In this paper speed-ups of 1.05-13.88 and 1.34-5.54 were observed for using hand optimized SIMD intrinsic functions rather than gcc compiler auto-vectorization for ARM and Intel platforms respectively. Preliminary analysis of the assembly code for the conversion benchmark showed that auto-vectorization required more instructions per pixel than intrinsics, failing to treat multiple pixels at a time. Extending this analysis to the other benchmarks and investigating whether other compilers are better able to auto-vectorize these codes is of interest. For each benchmark as the gcc compiled hand optimized intrinsic code and the auto-vectorized code was the same across all the Intel platforms and across all the ARM platforms, similar AUTO:HAND speedup ratios might be expected within each group of processors (with possible differences between the Intel and the ARM processor groups). Our results showed that this is not always the case. This is presumably due to low level hardware implementation details. Understanding the cause of these differences requires further work.

Most of the platforms used in our experiments might be considered to be “low power”. Extending the results reported here to include power efficiency in terms of performance per watt is of interest. This is particularly pertinent as we seek to determine which processors are most likely to serve as building blocks for exascale machines. In this respect extending our experiments to include the new ARM Cortex-A15 with improved pipelining and power management would also be of interest.

VII. Acknowledgements

This work is supported in part by the Australian Research Council Discovery Project DP0987773. We thank Kurt L. Keville at the Massachusetts Institute of Technology, for providing us with access to the ODROID-X development platform and Nicólas Erdödy of Open Parallel for providing access to a Samsung Galaxy S3 smart-phone.

References

[1] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano, "SPARC64 VIIIFX: A New-Generation Octocore Processor for Petascale Computing," *Micro, IEEE*, vol. 30, no. 2, pp. 30–40, 2010.

[2] M. Gschwind, "Blue Gene/Q: design for sustained multi-petaflop computing," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 245–246.

[3] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementationa performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[4] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *International Symposium on Memory Management: Proceedings of the 6 th international symposium on Memory management*, vol. 21, no. 22, 2007, pp. 103–104.

[5] ARM Limited, *Introducing NEON™ - Development Article*, 2009.

[6] ARM Limited, *ARM v8-A Instruction Set Overview*, July 2012.

[7] J. Dongarra and P. Luszczek, "Anatomy of a Globally Recursive Embedded LINPACK Benchmark," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2012.

[8] K. Cheng and Y. Wang, "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*. IEEE, 2011, pp. 1–4.

[9] M. Ibrahim, M. Rupp, and H. Fahmy, "Code transformations and SIMD impact on embedded software energy/power consumption," in *Computer Engineering & Systems, 2009. ICCES 2009. International Conference on*. IEEE, 2009, pp. 27–32.

[10] Intel Corporation, "Intel C++ Intrinsic Reference," 2007, Accessed: 29/11/2012.

[11] S. Maleki, Y. Gao, M. Garzarán, T. Wong, and D. Padua, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[12] ARM Limited, *ARM Architecture Reference Manual - ARM v7-A and ARM v7-R edition Errata markup*, 2011.

[13] S. Bentmar Holgersson, "Optimising IIR Filters Using ARM NEON," 2012.

[14] R. Kutil, "Parallelization of IIR filters using SIMD extensions," *2008 15th International Conference on Systems, Signals and Image Processing*, vol. 1, no. June, pp. 65–68, Jun. 2008.

[15] J. Wan, R. Wang, H. Lv, L. Zhang, W. Wang, C. Gu, Q. Zheng, and W. Gao, "AVS video decoding acceleration on ARM Cortex-A with NEON," in *Signal Processing, Communication and Computing (ICSPCC), 2012 IEEE International Conference on*. IEEE, 2012, pp. 290–294.

[16] T. Rintaluoma, "Optimizing H. 264 Decoder for Cortex-A8 with ARM NEON OpenMax DL Implementation," *IQ Magazine: The Smart Approach to Designing with the ARM Architecture*, vol. 8, no. 2, pp. 32–37, 2009.

[17] T. Rintaluoma and O. Silven, "SIMD performance in software based mobile video coding," *Embedded Computer System*, pp. 79–85, Jul. 2010.

[18] A. Blake, "Computing the fast Fourier transform on SIMD microprocessors," Ph.D. dissertation, University of Waikato, 2012.

[19] P. Gepner, V. Gamayunov, and D. Fraser, "Early performance evaluation of AVX for HPC," *Procedia Computer Science*, vol. 4, pp. 452–460, 2011.

[20] H. Jeong, S. Kim, W. Lee, and S.-H. Myung, "Performance of SSE and AVX Instruction Sets," *arXiv preprint arXiv:1211.0820*, 2012.

[21] A. Heinecke and D. Pflüger, "Multi-and many-core data mining with adaptive sparse grids," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 2011, p. 29.

[22] A. Heinecke, M. Klemm, H. Pabst, and D. Pflüger, "Towards high-performance implementations of a custom HPC kernel using® array building blocks," *Facing the Multicore-Challenge II*, pp. 36–47, 2012.

[23] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with OpenCV," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[24] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. Cambridge, MA: O'Reilly, 2008.

[25] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[26] Intel Corporation, "Intel Atom™ Processors N450 , D410 and D510 for Embedded Computing," 2010, accessed: 14/10/2012.

[27] Texas Instruments, "DM3730, DM3725 Digital Media Processors," Tech. Rep. July, 2011.

[28] "Hardkernel ODROID," http://www.hardkernel.com/renewal_2011/main.php, Tech. Rep., 2013.

[29] "OpenCV For Android," http://code.opencv.org/projects/opencv/wiki/OpenCV4Android, accessed: 14/10/2012.

[30] "Android NDK," http://developer.android.com/tools/sdk/ndk/index.html, accessed: 14/10/2012.