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

Gaurav Mitra[1] Beau Johnston[1] Alistair P. Rendell[1] Eric McCreath[1] Jun Zhou[2]

[1]Research School of Computer Science,
Australian National University,
Canberra, Australia

[2]School of Information and Communication Technology,
Griffith University, Nathan, Australia

May 20, 2013

Outline

Outline

Australian National University

Motivation: Energy

Problem?

- Energy consumption: Major roadblock for future exascale systems
- Astronomical increase in TCO

Heterogeneous Systems Widely Used. Top 3 on Green500 (NOV, 2012):

1. Beacon - Appro GreenBlade GB824M (**2.499** GFLOPS/Watt):
   - Intel Xeon Phi 5110P Many-Integrated-Core (MIC)
2. SANAM - Adtech ESC4000/FDR G2 (**2.351** GFLOPS/Watt):
   - AMD FirePro S10000
3. Titan - Cray XK7 (**2.142** GFLOPS/Watt):
   - NVIDIA K20x

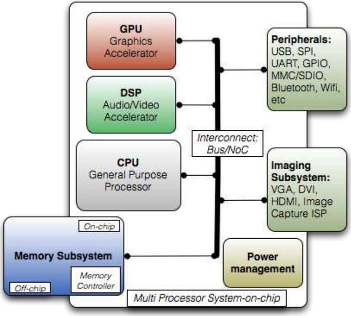
(a) Xeon Phi

(b) AMD Firepro

(c) Tesla K20

Motivation: ARM System-on-Chips

Australian National University



- J. Dongarra measured 4 GFLOPS/Watt from Dual-core ARM Cortex-A9 CPU in an Apple Ipad 2[a]. Proposed three tier categorization:
  - 1 GFLOPS/Watt: Desktop and server processors
  - 2 GFLOPS/Watt: GPU Accelerators
  - 4 GFLOPS/Watt: ARM Cortex-A processors
- Primarily used ARM VFPv3 Assembly Instructions from a High Level Python Interface
- ARM NEON SIMD operations not used
- On-chip GPU not used

[a] Jack Dongarra and Piotr Luszczek. "Anatomy of a Globally Recursive Embedded LINPACK Benchmark". In: *IEEE High Performance Extreme Computing Conference (HPEC)* (2012).

Primary Research Questions

Australian National University

1. How can the underlying hardware on ARM SoCs be effectively exploited?
    1. Full utilization of multi-core CPU with FPU and SIMD units
    2. Dispatch *Data Parallel* or *Thread Parallel* sections to on-chip Accelerators
2. Can this be automated? If so, how?
3. What performance can be achieved for message passing (MPI) between nodes on an ARM SoC cluster?
4. What level of energy efficiency can be achieved?

We focus on Step 1.1 and exploiting SIMD units in this work.

Outline

SIMD Extentions in CISC and RISC alike

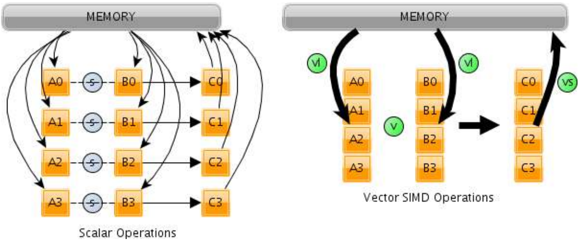
Origin:

- The Cray-1 @ 80 MHz at Los Alamos National Lab, 1976
- Introduced CPU registers for SIMD vector operations
- 250 MFLOPS when SIMD operations utilized effectively

Extensive use of SIMD extensions in Contemporary HPC Hardware:

- Complex Instruction Set Computers (CISC)
  - Intel Streaming SIMD Extensions (SSE): 128-bit wide XMM registers
  - Intel Advanced Vector Extensions (AVX): 256-bit wide YMM registers
- Reduced Instruction Set Computers (RISC)
  - SPARC64 VIIIFX (HPC-ACE): 128-bit registers
  - PowerPC A2 (Altivec, VSX): 128-bit registers
- Single Instruction Multiple Thread (SIMT): GPUs

SIMD Operations Explained



- Scalar: 8 loads + 4 scalar adds + 4 stores = 16 ops
- Vector: 2 loads + 1 vector add + 1 store = 4 ops
- Speedup: $16/4 = 4\times$
- Simple expression of Data Level Parallelism

Using SIMD Operations

Australian National University

1. Assembly:

```
.text
.arm
.global double_elements
double_elements:
vadd.i32 q0,q0,q0
bx
lr
.end
```

2. Compiler Intrinsic Functions:

```
#include <arm_neon.h>
uint32x4_t double_elements(uint32x4_t input)
{
    return(vaddq_u32(input, input));
}
```

3. Compiler Auto-vectorization:

```
unsigned int* double_elements(unsigned int* input, int len)
{
    int i;
    for(i = 0; i < len; i++)
        input[i] += input[i]

    return input;
}
```

Outline

Australian National University

Objective

How effective are ARM NEON operations compared to Intel SSE?

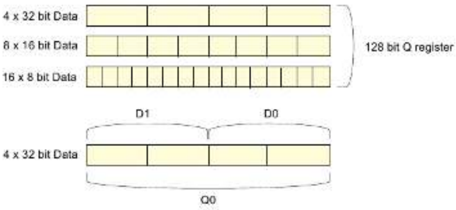
- *Effectiveness* measured in terms of relative Speed-ups
- Evaluation of ability of NEON and SSE to accelerate real-world application codes

What is the optimal way to utilize NEON and SSE operations without writing assembly? We compare:

- Compiler Intrinsics
- Compiler Auto-vectorization

ARM NEON Registers

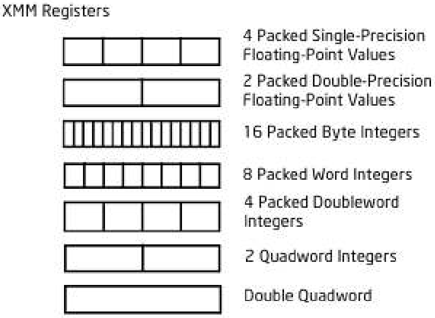
Australian National University



- ARM Advanced SIMD (NEON)
- 32 64-bit Registers
- Shared by VFPv3 instructions
- NEON views:
  - Q0-Q15: 16 128-bit Quad-word
  - D0-D31: 32 64-bit Double-word
- 8, 16, 32, 64-bit Integers
- ARMv7: 32-bit SP Floating-point
- ARMv8: 32-bit SP & 64-bit DP

Intel SSE Registers

Australian National University



- Intel Streaming SIMD Extensions (SSE)
- 8 128-bit XMM Registers
- XMM0 - XMM7
- 8, 16, 32, 64-bit Integers
- 32-bit SP & 64-bit DP

OpenCV

Open Computer Vision (OpenCV) library:

- Image processing routines
- Contains $\geq$ 400 commonly used operations
- Written in C++
- Major modules:
  - *Core*: Basic data structures and functions used by all other modules. Matrix operations, vector arithmetic, data type conversions etc.
  - *Imgproc*: Higher level image processing ops such as filters

Which routines to test?

- OpenCV 2.4.3: **187 SSE2 Intrinsic Optimized Functions** in 55 files
- OpenCV 2.4.3: **6 NEON Intrinsic Optimized Functions** in 3 files

Analogous to existing SSE2 Functions, NEON Functions were written.

OpenCV: Element Wise Operations

Core: (1) *Conversion* of 32-bit float to 16-bit short int:

Algorithm 1 Pseudocode: Cast Each Pixel

for all pixels in Image **do**
  Saturate-Cast-F32-to-S16(pixel)
end for

Imgproc: (2) *Binary thresholding* each pixel:

Algorithm 2 Pseudocode: Binary Threshold

for all pixels in Image **do**
  if pixel $\leq$ threshold **then**
    pixel $\leftarrow$ threshold
  else
    pixel $\leftarrow$ pixel
  end if
end for

OpenCV: Convolution (Filter) Operations

Imgproc: (3) *Gaussian blur* & (4) *Sobel filter*:

Algorithm 3 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

Combined Operation: (5) *Edge Detection* (Sobel Filter + Binary Threshold)

Platforms: ARM

Australian National University

(a) Samsung Nexus S (Exynos 3110: 1-Cortex-A8, 1Ghz)

(b) Samsung Galaxy Nexus (TI OMAP 4460: 2-Cortex-A9, 1.2Ghz)

(c) Samsung Galaxy S3 (Exynos 4412: 4-Cortex-A9, 1.4Ghz)

(d) Gumstix Overo Firestorm (TI DM 3730: 1-Cortex-A8, 0.8Ghz)

(e) Hardkernel ODROID-X (Exynos 4412: 4-Cortex-A9, 1.3Ghz)

(f) NVIDIA CARMA DevKit (Tegra T30: 4-Cortex-A9, 1.3Ghz)

Platforms: Intel

(a) Intel Atom D510 (2 cores, 4 threads, 1.66Ghz)

(b) Intel Core 2 Quad Q9400 (4 cores, 4 threads, 2.66Ghz)

(c) Intel Core i7 2820QM (4 cores, 8 threads, 2.3Ghz)

(d) Intel Core i5 3360M (2 cores, 4 threads, 2.8Ghz)

Platforms: ARM and Intel

Australian National University

PROCESSOR	CODENAME	Launched	Threads/Cores/Ghz	Cache L1/L2/L3 (KB)	Memory	SIMD Extensions
			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
			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

Table: Platforms Used in Benchmarks

Platforms: ARM and Intel

PROCESSOR	CODENAME	Launched	Threads/Cores/Ghz	Cache L1/L2/L3 (KB)	Memory	SIMD Extensions
			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
			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

Table: Platforms Used in Benchmarks

Code, Compilers and Tools

- Linux platforms:
  - OpenCV 2.4.2 optimized source
  - Benchmarks written in C++
  - CMake cross-compiler toolchain
  - GCC 4.6.3 for both Intel and ARM
  - Intel opts: -O3 -msse -msse2
  - ARM opts: -mfpu=neon -ftree-vectorize -mtune=cortex-a8/a9 -mfloat-abi=softfp/hard
- Android Smart-phones:
  - OpenCV4Android with OpenCV 2.4.2 optimized source
  - Android NDK r8b compiler - GCC 4.6.x

Methodology

- Two versions of OpenCV compiled:
  - HAND: Intrinsics + Auto-vectorization
    - cv::setUseOptimized(bool on)
  - AUTO: Auto-vectorization Only
    - cv::setUseOptimized(bool off)
- Relative speedups:
  - Intel HAND vs. Intel AUTO
  - ARM HAND vs. ARM AUTO
- Both versions benchmarked on different image sizes
  - 640×480: 0.3 Mpx, 1.2MB
  - 1280×960: 1 Mpx, 4.7MB
  - 2560×1920: 5 Mpx, 19MB
  - 3264×2448: 8 Mpx, 23MB
- Cycled through 5 different images of each resolution 25 times, over 100 runs of a benchmark
- High resolution timer with accuracy $> 10^{-6}$ was used

Outline

Australian National University

Results: Convert Float to Short

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

Results: Convert Float to Short

Australian National University

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

Results: Convert Float to Short

Australian National University

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

Results: Convert Float to Short

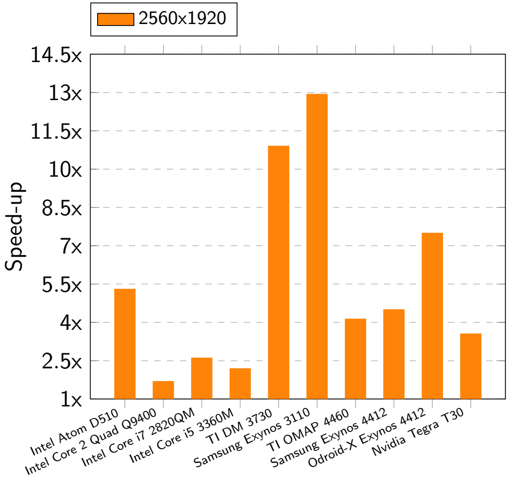


Figure: Convert Float to Short relative speed-up factors

Analysis: Convert Float to Short

Algorithm in C:

```
for( ; x < size.width; x++ )
{
    dst[x] = saturate_cast<short>(src[x]);
}

template<> inline short saturate_cast<short>(float v)
{
    int iv = cvRound(v);
    return saturate_cast<short>(iv);
}

CV_INLINE int cvRound( double value )
{
    return (int)(value + (value >= 0 ? 0.5 : -0.5));
}

template<> inline short saturate_cast<short>(int v)
{
    return (short)((unsigned)(v - SHRT_MIN) <= (unsigned)USHRT_MAX ?
    v : v > 0 ? SHRT_MAX : SHRT_MIN);
}
```

Analysis: Convert Float to Short

Australian National University

Using NEON and SSE2 Intrinsics:

```
/* 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);
}
/* 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);
}
```

Analysis: Convert Float to Short

NEON Assembly:

14 Operations (8 pixels at a time):

```
/* 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

vld1.32 {d16-d17}, [r2]!
cmp r3, fp
vcvt.s32.f32 q8, q8
vld1.32 {d18-d19}, [r2]
vcvt.s32.f32 q9, q9
vqmovn.s32 d16, q8
vqmovn.s32 d18, q9
vorr d17, d18, d18
vst1.16 {d16-d17}, [r0]

bne.n 48 <cv::cvt32f16s(
     float const*,
     unsigned int,
     unsigned char const
     *, unsigned int,
     short*, unsigned int
     , cv::Size_<int>,
     double*)+0x48>
```

16 Operations (1 pixel at a time):

```
/* Auto-vectorized ARM Assembly */
8e: vldmia r6!, {s15}
vcvt.f64.f32 d16, s15
vmov r0, r1, d16
bl 0 <lrint>
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>
```

Results: Binary Threshold

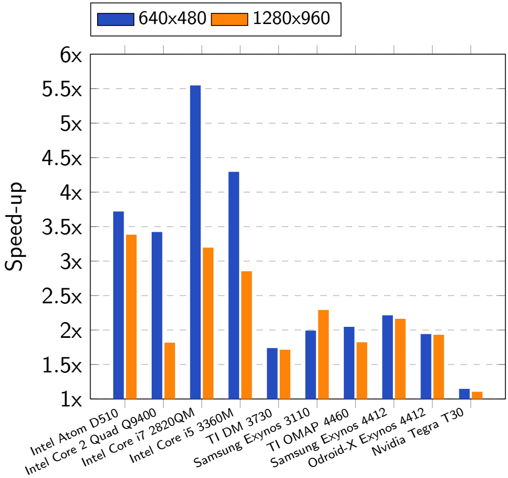
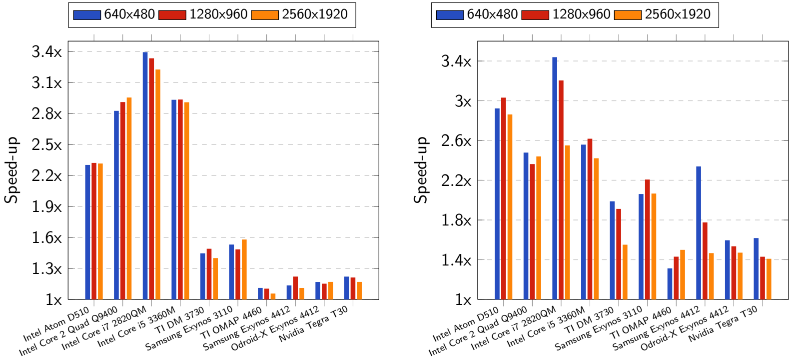


Figure: Binary Image Thresholding relative speed-up

Results: Convolution Operations

Australian National University



(a) Gaussian Blur (b) Sobel Filter

Figure: Convolution Operation relative speed-up factors

Results: Edge Detection

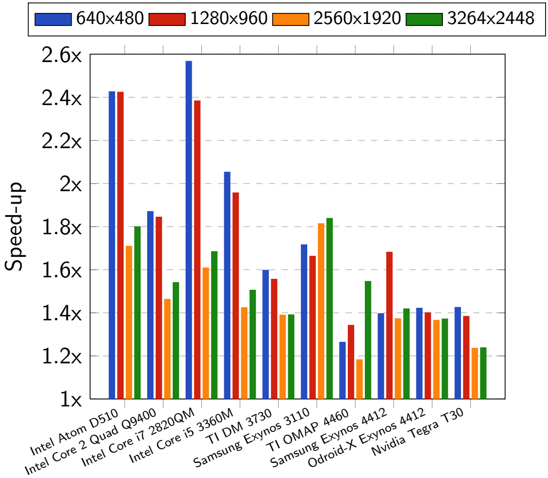


Figure: Edge Detection relative speed-up factors

Observations

- Intrinsic functions consistently provide speed-up compared to GCC auto-vectorization
  - AUTO required more instructions/pixel than HAND
  - Non-aligned memory operations done by AUTO
- ARM NEON operations provide higher speed-up for element-wise operations compared to convolution operations
  - More instructions/pixel required for convolutions
- Within a given processor type, the results were very similar for all image sizes, with some exceptions for 0.3 and 1 Mpx cases

Observations

- AUTO absolute times on Android platforms significantly better than AUTO absolute times on ARM Linux platforms
    - Android optimized linux kernel
    - BIONIC *libc* on Android (no C++ Exceptions, other optimizations)
- ODROID-X consistently outperforms the Tegra-T30 while both have 1.3Ghz ARM Cortex-A9 cores
    - *libc* using software floating point emulation (soft float) on Tegra T30
- Low-level hardware implementation differences (latencies, pipelines etc) amongst Intel platforms and amongst ARM platforms lead to unexpected AUTO:HAND speed-up ratios

Outline

Australian National University

Revisiting Initial Objectives

Motivating Research Question and Objectives:

1. How can the underlying hardware on ARM SoCs be effectively exploited?
   1. Full utilization of multi-core CPU with FPU and SIMD units
   2. Dispatch *Data Parallel* or *Thread Parallel* sections to on-chip Accelerators

Step 1.1: Objectives:

1. How effective are NEON operations compared to Intel SSE?
2. Evaluation of ability NEON and SSE to accelerate real-world application codes
3. What is the optimal way to utilize NEON and SSE operations without writing assembly?

For a Single Core and its SIMD unit, we found:

- ARM NEON provides comparable speed-ups to Intel SSE2
- Compiler intrinsic functions are optimal compared to auto-vectorization
- Speed-ups between 1.05-13.88 for real-world HPC application codes across both ARM and Intel platforms were observed

Future Work

Australian National University

- Extension of results to include energy efficiency
- Utilization of all cores and SIMD units on each core
- Further Evaluation of ARM Cortex-A15 vs. A9 and A8

Thank you!