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

Gaurav Mitra\textsuperscript{1} Beau Johnston\textsuperscript{1} Alistair P. Rendell\textsuperscript{1} Eric McCreath\textsuperscript{1} Jun Zhou\textsuperscript{2}

\textsuperscript{1}Research School of Computer Science, Australian National University, Canberra, Australia

\textsuperscript{2}School of Information and Communication Technology, Griffith University, Nathan, Australia

May 20, 2013
Outline

1. Motivation
2. Single Instruction Multiple Data (SIMD) Operations
3. Use of SIMD Vector Operations on ARM and Intel Platforms
4. Results & Observations
5. Conclusion
Outline

1. Motivation
   - Energy and Heterogeneity
   - ARM System-on-chips: A viable alternative

2. Single Instruction Multiple Data (SIMD) Operations

3. Use of SIMD Vector Operations on ARM and Intel Platforms

4. Results & Observations

5. Conclusion
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):

   - 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

- 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

Primary Research Questions

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

1 Motivation

2 Single Instruction Multiple Data (SIMD) Operations
   - SIMD CPU Extensions
   - Understanding SIMD Operations
   - Using SIMD Operations

3 Use of SIMD Vector Operations on ARM and Intel Platforms

4 Results & Observations

5 Conclusion
Single Instruction Multiple Data (SIMD) Operations

SIMD Extensions 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

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

1 Motivation

2 Single Instruction Multiple Data (SIMD) Operations

3 Use of SIMD Vector Operations on ARM and Intel Platforms
   - Processor Registers
   - The OpenCV Library
   - OpenCV routines benchmarked
   - Platforms Evaluated
   - Experimental Methodology

4 Results & Observations

5 Conclusion
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

- 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

- 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 ≥ 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

```plaintext
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

```plaintext
for all pixels in Image do 
    if pixel \leq\ threshold then 
        pixel ← threshold
    else 
        pixel ← 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 $I_{(\ast, \ast)} += I_{(x, y)} \times S_{(x, y)}$
        end for
    end for
end for

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

(a) Samsung Nexus S (Exynos 3110: 1-Cortex-A8, 1Ghz)
(b) Samsung Galaxy Nexus (TI OMAP 4460: 4412: 4-Cortex-A9, 1.4Ghz)
   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

<table>
<thead>
<tr>
<th>PROCESSOR</th>
<th>CODENAME</th>
<th>Launched</th>
<th>Threads/Cores/Ghz</th>
<th>Cache L1/L2/L3 (KB)</th>
<th>Memory</th>
<th>SIMD Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TI DM 3730</td>
<td>DaVinci</td>
<td>Q2’10</td>
<td>1/1.ARM Cortex-A8/0.8</td>
<td>32(I,D)/256/ No L3</td>
<td>512MB DDR</td>
<td>VFPv3/NEON</td>
</tr>
<tr>
<td>Samsung Exynos 3110</td>
<td>Exynos 3 Single</td>
<td>Q1’11</td>
<td>1/1.ARM Cortex-A8/1.0</td>
<td>32(I,D)/512/ No L3</td>
<td>512MB LPDDR</td>
<td>VFPv3/NEON</td>
</tr>
<tr>
<td>TI OMAP 4460</td>
<td>Omp</td>
<td>Q1’11</td>
<td>2/2.ARM Cortex-A9/1.2</td>
<td>32(I,D)/1024/ No L3</td>
<td>1GB LPDDR2</td>
<td>VPFPv3/NEON</td>
</tr>
<tr>
<td>Samsung Exynos 4412</td>
<td>Exynos 4 Quad</td>
<td>Q1’12</td>
<td>4/4.ARM Cortex-A9/1.4</td>
<td>32(I,D)/1024/ No L3</td>
<td>1GB LPDDR2</td>
<td>VPFPv3/NEON</td>
</tr>
<tr>
<td>Samsung Exynos 4412</td>
<td>ODROID-X</td>
<td>Q2’12</td>
<td>4/4.ARM Cortex-A9/1.3</td>
<td>32(I,D)/1024/ No L3</td>
<td>1GB LPDDR2</td>
<td>VPFPv3/NEON</td>
</tr>
<tr>
<td>NVIDIA Tegra T30</td>
<td>Tegra 3, Kal-El</td>
<td>Q1’11</td>
<td>4/4.ARM Cortex-A9/1.3</td>
<td>32(I,D)/1024/ No L3</td>
<td>2GB DDR3L</td>
<td>VPFPv3/NEON</td>
</tr>
<tr>
<td><strong>INTEL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Atom D510</td>
<td>Pineview</td>
<td>Q1’10</td>
<td>4/2/1.66</td>
<td>32(I),24(D)/1024/ No L3</td>
<td>4GB DDR2</td>
<td>SSE2/SSE3</td>
</tr>
<tr>
<td>Intel Core 2 Quad Q9400</td>
<td>YorkField</td>
<td>Q3’08</td>
<td>4/4/2.66</td>
<td>32(I,D)/3072/ No L3</td>
<td>8GB DDR3</td>
<td>SSE*</td>
</tr>
<tr>
<td>Intel Core i7 2820QM</td>
<td>Sandy Bridge</td>
<td>Q1’11</td>
<td>8/4/2.3</td>
<td>32(I,D)/256/8192</td>
<td>8GB DDR3</td>
<td>SSE*/AVX</td>
</tr>
<tr>
<td>Intel Core i5 3360M</td>
<td>Ivy Bridge</td>
<td>Q2’12</td>
<td>4/2/2.8</td>
<td>32(I,D)/256/3072</td>
<td>16GB DDR3</td>
<td>SSE*/AVX</td>
</tr>
</tbody>
</table>

**Table:** Platforms Used in Benchmarks
Platforms: ARM and Intel

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

**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 x 480: 0.3 Mpx, 1.2MB
  - 1280 x 960: 1 Mpx, 4.7MB
  - 2560 x 1920: 5 Mpx, 19MB
  - 3264 x 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

1. Motivation

2. Single Instruction Multiple Data (SIMD) Operations

3. Use of SIMD Vector Operations on ARM and Intel Platforms

4. Results & Observations
   - Results: Convert Float to Short
   - Analysis: Convert Float to Short
   - Results: Binary Threshold
   - Results: Convolutions
   - Observations

5. Conclusion
### Results: Convert Float to Short

<table>
<thead>
<tr>
<th>SIMD</th>
<th>INTEL (SSE2)</th>
<th>ARM (NEON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Atom D510</td>
<td>TI DM 3730</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Core 2 Q9400</td>
<td>Exynos 3110</td>
</tr>
<tr>
<td>Image Size</td>
<td>Core i7 2820QM</td>
<td>TI OMAP 4460</td>
</tr>
<tr>
<td>Optimized</td>
<td>Core i5 3360M</td>
<td>Exynos 4412</td>
</tr>
<tr>
<td>640x480</td>
<td>Auto 0.01492 0.00182</td>
<td>Odroid-X Ex-4412</td>
</tr>
<tr>
<td>HAND</td>
<td>0.00283 0.00136</td>
<td>Tegra T30</td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.27 1.34 2.93</td>
<td></td>
</tr>
<tr>
<td>1280x960</td>
<td>Auto 0.05952 0.00711</td>
<td>0.01422</td>
</tr>
<tr>
<td>HAND</td>
<td>0.01129 0.00436</td>
<td></td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.27 1.63 2.73</td>
<td></td>
</tr>
<tr>
<td>2560x1920</td>
<td>Auto 0.23770 0.02845</td>
<td>0.01422</td>
</tr>
<tr>
<td>HAND</td>
<td>0.04472 0.01670</td>
<td></td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.32 1.70 2.62</td>
<td></td>
</tr>
<tr>
<td>3264x2448</td>
<td>Auto 0.43863 0.06392</td>
<td>0.01422</td>
</tr>
<tr>
<td>HAND</td>
<td>0.12374 0.03702</td>
<td></td>
</tr>
<tr>
<td>Speed-up</td>
<td>3.54 1.73 2.33</td>
<td></td>
</tr>
</tbody>
</table>

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

---

Mitra et. al. (ANU, Griffith)
### Results: Convert Float to Short

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

<table>
<thead>
<tr>
<th>SIMD</th>
<th>INTEL (SSE2)</th>
<th>ARM (NEON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Atom D510</td>
<td>Core 2 Q9400</td>
</tr>
<tr>
<td>Image Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>AUTO</td>
<td>0.01492</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.00283</td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.27</td>
<td>1.34</td>
</tr>
<tr>
<td>1280x960</td>
<td>AUTO</td>
<td>0.05952</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.01129</td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.27</td>
<td>1.63</td>
</tr>
<tr>
<td>2560x1920</td>
<td>AUTO</td>
<td>0.23770</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.04472</td>
</tr>
<tr>
<td>Speed-up</td>
<td>5.32</td>
<td>1.70</td>
</tr>
<tr>
<td>3264x2448</td>
<td>AUTO</td>
<td>0.43863</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.12374</td>
</tr>
<tr>
<td>Speed-up</td>
<td>3.54</td>
<td>1.73</td>
</tr>
</tbody>
</table>

**Note:** The data represents the time taken to convert a Float to a Short Int on various processors for different image sizes. The fastest time is highlighted in orange.
## Results: Convert Float to Short

<table>
<thead>
<tr>
<th>SIMD</th>
<th>INTEL (SSE2)</th>
<th>ARM (NEON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Atom D510</td>
<td>Core 2 Q9400</td>
</tr>
<tr>
<td>Image Size</td>
<td>Intrinsic Optimized</td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>AUTO</td>
<td>0.01492</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.00283</td>
</tr>
<tr>
<td><strong>Speed-up</strong></td>
<td>5.27</td>
<td>1.34</td>
</tr>
<tr>
<td>1280x960</td>
<td>AUTO</td>
<td>0.05952</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.01129</td>
</tr>
<tr>
<td><strong>Speed-up</strong></td>
<td>5.27</td>
<td>1.63</td>
</tr>
<tr>
<td>2560x1920</td>
<td>AUTO</td>
<td>0.23770</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.04472</td>
</tr>
<tr>
<td><strong>Speed-up</strong></td>
<td>5.32</td>
<td>1.70</td>
</tr>
<tr>
<td>3264x2448</td>
<td>AUTO</td>
<td>0.43863</td>
</tr>
<tr>
<td></td>
<td>HAND</td>
<td>0.12374</td>
</tr>
<tr>
<td><strong>Speed-up</strong></td>
<td>3.54</td>
<td>1.73</td>
</tr>
</tbody>
</table>

**Table:** Time (in seconds) to perform conversion of Float to Short Int
Results & Observations

Results: Convert Float to Short

Figure: Convert Float to Short relative speed-up factors
Analysis: Convert Float to Short

Algorithm in C:

```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);
}
```
Using NEON and SSE2 Intrinsics:

```c
/* 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
3     add.w r0, r9, r3  #x+8
5     adds r3, #16  #src+x
7     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]
```

16 Operations (1 pixel at a time):

```
/* Auto-vectorized ARM Assembly */
8e:   vldmia r6!, {s15}
3     vcvt.f64.f32 d16, s15
5     vmov r0, r1, d16
7     b1 0 <lrint>
9     cmp r2, r8
9     bls.n b2 <cv::cvt32f16s( float const*,
             unsigned int, unsigned char const*,
             unsigned int, short*, unsigned int
             , cv::Size_<int>, double*)+0xb2>

11    cmp r0, #0
13    ite gt
15    movgt r3, sl
17    strh.w r3, #32768 ; 0x8000
19    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

(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

1. Motivation
2. Single Instruction Multiple Data (SIMD) Operations
3. Use of SIMD Vector Operations on ARM and Intel Platforms
4. Results & Observations
5. Conclusion
   - Re-visiting objectives
   - Future Work
Revisiting Initial Objectives

Motivating Research Question and Objectives:

1. How can the underlying hardware on ARM SoCs be effectively exploited?
   - Full utilization of multi-core CPU with FPU and SIMD units
   - 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

- 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!