# OpenCL Framework for ARM Processors with NEON Support

Gangwon Jo
Department of Computer Science and Engineering
Seoul National University
gangwon@aces.snu.ac.kr

Won Jong Jeon
Samsung Research America - Silicon Valley
won.jeon@samsung.com

Wookeun Jung
Department of Computer Science and Engineering
Seoul National University
wookeun@aces.snu.ac.kr

Gordon Taft
Samsung Research America - Silicon Valley
gordon.taft@samsung.com

Jaejin Lee
Department of Computer Science and Engineering
Seoul National University
jlee@cse.snu.ac.kr

## Abstract

The state-of-the-art ARM processors provide multiple cores and SIMD instructions. OpenCL is a promising programming model for utilizing such parallel processing capability because of its SPMD programming model and built-in vector support. Moreover, it provides portability between multicore ARM processors and accelerators in embedded systems. In this paper, we introduce the design and implementation of an efficient OpenCL framework for multicore ARM processors. Computational tasks in a program are implemented as OpenCL kernels and run on all CPU cores in parallel by our OpenCL framework. Vector operations and built-in functions in OpenCL kernels are optimized using the NEON SIMD instruction set. We evaluate our OpenCL framework using 37 benchmark applications. The result shows that our approach is effective and promising.

***Categories and Subject Descriptors*** D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Code generation, Compilers, Optimizations, Run-time environments

***General Terms*** Design, Experimentation, Languages, Measurement, Performance

***Keywords*** Embedded System, Multicore, OpenCL, SIMD

## 1. Introduction

ARM processors are widely used for embedded systems including smartphones, tablets, netbooks, and so on. The state-of-the-art ARM processors (e.g., ARM Cortex-A9 and Cortex-A15 MPCore) have multiple cores (dual- or quad-core) and support a 64- and 128-bit SIMD instruction set (i.e., NEON[6] instructions). Compute-intensive applications such as 3D gaming, physics simulation, image processing, and augmented reality may enhance the performance by utilizing the parallel processing capability of the ARM processors. To achieve this, an easy and efficient parallel programming model is required.

OpenCL (Open Computing Language)[16] is an open standard for general purpose parallel programming developed by the Khronos Group. It mainly targets accelerators in heterogeneous computing systems, such as GPUs[4, 20], Intel Xeon Phi coprocessors[12], Cell BE processors[18], and FPGAs[3]. Compute-intensive and data-parallel tasks in a program are implemented as SPMD-threaded OpenCL kernels using the OpenCL C programming language that is similar to C99. Then, they can be offloaded to many processing elements within an accelerator. However, OpenCL can also be used as a programming model for multicore CPUs[10, 12, 18]. An OpenCL kernel is not offloaded, but run on multiple CPU cores in parallel.

There are some reasons why OpenCL is promising for parallel programming in multicore ARM processors. First, because of the SPMD nature of OpenCL kernels, the OpenCL programming model fits data-parallel programs well. Heavy computational workloads in embedded systems usually have the data parallelism. Second, OpenCL C provides built-in vector types and operators. It enables programmers to easily vectorize OpenCL kernels by hand. The OpenCL C compiler can translate a vector operation to one or more NEON instructions. The result may outperform the auto-vectorized code generated by optimizing compilers. Third, and the most important of all, OpenCL provides portability across different types of processors. Many embedded system-on-chips (SoCs) such as Samsung Exynos, NVIDIA Tegra, and TI OMAP contain both an ARM processor and an accelerator.


*WPMVP '14*, February 16, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2568058.2568064

Using OpenCL frameworks for accelerators (e.g., OpenCL SDK for ARM Mali GPUs[5]), an OpenCL application can also run on the accelerators without any modification.

In this paper, we present an OpenCL framework for multicore ARM processors. The prior studies proposed methods to compile and execute applications written in OpenCL[10, 12, 18] or other accelerator programming models such as CUDA[11, 15, 24] on multicore CPUs. But they target x86 processors, while our work focuses on ARM processors for embedded systems. Our OpenCL framework is composed of an *OpenCL runtime* and an *OpenCL C compiler*. The OpenCL runtime schedules commands issued by the OpenCL applications, manages memory objects, and executes OpenCL kernels on all CPU cores in parallel. The OpenCL C compiler transforms an OpenCL kernel to the ARM binary. It adopts the work-item coalescing technique[18] to execute multiple kernel instances on a single core efficiently. Also, it optimizes vector operations and built-in functions in OpenCL C using the NEON instruction set of ARM processors.

We implement our OpenCL framework and evaluate the performance using 37 benchmark applications from OpenCL samples in AMD APP SDK[4], Parboil benchmarks[25], and SNU NPB suite[22]. To the best of our knowledge, this is the first work to evaluate a rich set of OpenCL applications on a multicore embedded processor. The experimental results show that our OpenCL framework is efficient and OpenCL is a promising programming model for multicore ARM processors. The major contributions of this paper are the following:

- We introduce the design and implementation of an efficient OpenCL framework for multicore ARM processors. Especially, we propose the optimization techniques for OpenCL kernels that use NEON instructions, and implement them into our OpenCL C compiler.
- We evaluate our OpenCL framework with a quad-core ARM processor. From the experimental result, we show the effectiveness of the framework design and the NEON optimization techniques.
- We compare the performance of our OpenCL framework with that of another parallel programming model (i.e., OpenMP) and another OpenCL framework for ARM processors (i.e., PGCL[26]). We show that OpenCL applications combined with our OpenCL framework can achieve high performance on multicore ARM processors.

The rest of the paper is organized as follows. Section 2 describes the structure of our OpenCL framework, and Section 3 introduces the NEON optimization techniques in detail. Section 4 explains the target systems and benchmark applications used for the experiments. Section 5 shows and discusses the experimental results of our OpenCL framework. Section 6 describes the related work, and Section 7 concludes the paper.

## 2. The OpenCL Framework

This section describes the design and implementation of our OpenCL framework.

### 2.1 Programming Model and OpenCL C

The OpenCL standard defines OpenCL API functions and OpenCL C programming language. A host program is written in high-level programming languages for CPUs, such as C and C++. It uses OpenCL API functions to manage OpenCL objects and to enqueue commands to command-queues. When it enqueues a kernel execution command, it defines an index space called *NDRange*. An instance of the kernel is executed for each point in the NDRange. OpenCL C is used to write kernels. It is derived from the C99 specification, and adds some extensions and restrictions.

OpenCL C supports built-in vector types and operators. A vector type is defined with the type of elements (i.e., one of `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and `double`) followed by the number of elements $n$, where $n \in \{2, 3, 4, 8, 16\}$. Operators in C (e.g., `+`, `-`, `*`, and `/`) operate on built-in vector types. The operations are performed component-wise. OpenCL C also provides a rich set of built-in functions, including the following:

- Functions to query the global ID (`get_global_id(n)`), work-group ID (`get_group_id(n)`), and local ID (`get_local_id(n)`) of the current work-item. The argument `n` specifies the dimension which should be one of 0, 1, and 2.
- Functions for explicit type conversions. These functions are in the form of `convert_destType(x)`. For example, the `convert_int4(x)` function converts the argument `x` to an `int4` vector. The arguments can be either a scalar value or a vector with the size 4. If the argument is a scalar, it is replicated across all lanes of the vector.
- Tens of math functions such as `sin(x)`, `cos(x)`, and `exp(x)`. Each built-in math function may take a vector as an argument. In this case, the function operates component-wise and returns the result as a vector with the same size.
- Functions to read data from (or write data to) an image object. In a kernel, buffer objects are treated as arrays and can be accessed using the array subscript operator (`[]`). However, elements in an image object can only be read and written with the built-in functions.

### 2.2 OpenCL Platform on ARM Processors

Our OpenCL framework targets multicore ARM processors in embedded systems. Thus, the ARM processor acts as both a host processor and an OpenCL device. Since ARM processors have only a few cores, it is not beneficial to execute all work-items in an NDRange concurrently. This also causes significant context switching overhead. Instead, we adopt work-group level parallelism. Every core becomes a compute unit and, at the same time, a single processing element in the compute unit. Work-groups in the NDRange are distributed to the cores and executed in parallel. All

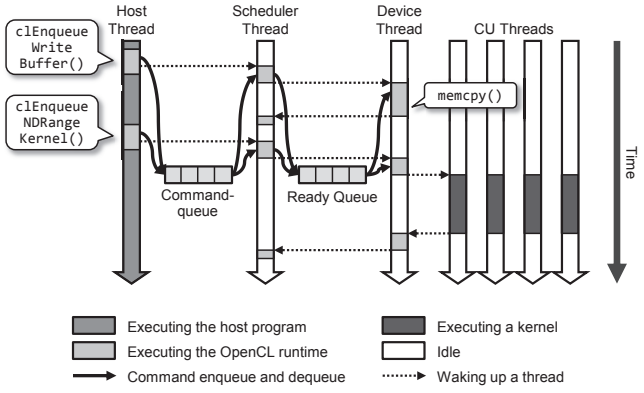


**Figure 1.** The OpenCL runtime.

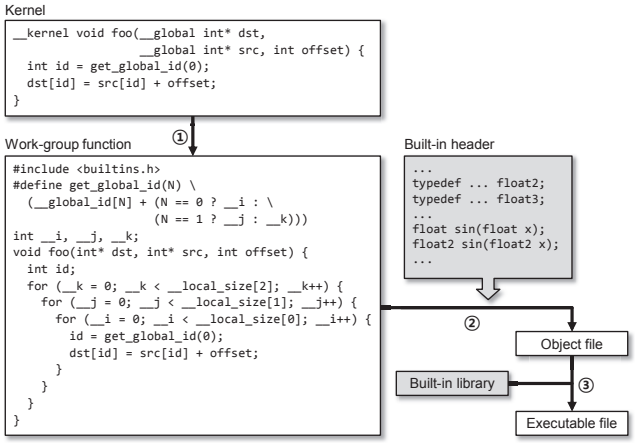


**Figure 2.** The compilation process of a kernel.

work-items in the same work-group are executed on the same core sequentially one by one.

A *host thread* executes the host program on a core. One of the cores should execute both the host program and work-groups assigned to it. OpenCL host programs usually enqueues commands to command-queues and just waits until the commands are completed. In this case, the overhead of the host thread is negligible. If the host program performs a compute-intensive task while a kernel is running, however, the host thread may degrade the performance of the kernel.

All memory regions of the OpenCL device (i.e., *global*, *constant*, *local* and *private*) are placed in the main memory. Buffer and image objects are created in the main memory using the `malloc()` function. Since the device memory is logically distinct from the main memory, however, the host program cannot access these objects directly. Instead, it should use memory commands (i.e., commands to copy data from/to the device memory).

### 2.3 OpenCL Runtime

Our OpenCL runtime provides an implementation of OpenCL API functions for the host program. To schedule and execute commands, the runtime creates a *scheduler thread*, a *device thread*, and a *CU thread* for each compute unit (i.e., each CPU core) besides the host thread. Figure 1 illustrates how these threads work. The host program enqueues commands to command-queues and defines the execution order between them using the OpenCL API functions. The scheduler thread dequeues commands that are *ready* to execute (i.e., all preceding commands are finished) from the command-queues. Then, it enqueues these commands to a *ready queue* that is shared with the device thread. The device thread fetches and executes commands in the ready queue. Memory commands are executed on the device thread using the `memcpy()` function. For a kernel execution command, the device thread evenly distributes work-groups in the NDRange to the CU threads and waits until all work-groups are completed. Each CU thread actually executes the assigned work-groups on a CPU core.

The scheduler thread goes to sleep after it checks all command-queues. When a new command is enqueued to a command-queue, the host thread wakes up the scheduler thread. In addition, the device thread wakes up the scheduler thread whenever it finishes the execution of a command. Similarly, the device thread sleeps if the ready queue is empty, and the scheduler thread wakes up the device thread after it enqueues a command to the command queue. As a result, our OpenCL runtime minimizes the overhead of the scheduler thread and the device thread. The dashed arrows in Figure 1 illustrates this mechanism.

### 2.4 OpenCL C Compiler

Figure 2 shows the compilation process of a kernel. Our OpenCL C compiler translates a kernel to a *work-group function* (①), and compiles it to an ARM binary (② and ③). The work-group function is a C++ function that executes all work-items in a single work-group sequentially. The CU threads in the OpenCL runtime call this function for each work-group.

There are two ways to implement the work-group function. Gummaraju *et al.*[10] call the `setjmp()` function at the end of a work-item to switch to the next work-item. On the other hand, Lee *et al.*[18] encloses the kernel code with a triply nested loop. If the kernel contains barriers, it is separated by barriers and each part is enclosed by a loop individually. This is called *work-item coalescing*. Our *source-to-source translator* adopts the work-item coalescing technique. Figure 2 shows an example of a work-group function. The array `__global_id` and `__local_size` indicates the global ID of the first work-item in the work-group and the size of the work-group, respectively. They are initialized by the CU thread before the function is called. The variables `__i`, `__j`, and `__k` indicate the local ID of the current work-item. The details of the work-item coalescing are omitted due to the page limit and can be found in the paper written by Lee *et al.*[18].

Our OpenCL C compiler uses a native compiler (i.e., GCC) for the steps ② and ③. Hence optimizations in GCC are naturally applied to kernels and boost the performance of the kernel. The work-group function may contain built-in types and functions of OpenCL C. To support them, the work-group function is compiled with a *built-in header* (②)

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

**Figure 3.** Using NEON intrinsics for GCC[6].

and then linked with a *built-in library* (③). The built-in header defines the built-in types of OpenCL C using structures and typedefs. It also declares the prototypes of the built-in functions. The built-in library provides their implementation. The compiler finally creates an executable file in the form of a shared library. Our OpenCL runtime uses dynamic loading to call the work-group functions in the shared library.

The source-to-source translator and built-in library included in an open-source OpenCL framework SnuCL[17] is used as the baseline of our OpenCL C compiler. We modify the source-to-source translator to support the caching mechanism and to eliminate the unnecessary overhead of work-group functions. We also implement the NEON optimizations described in Section 3 in the translator and built-in library.

## 3. Optimizations for NEON

NEON[6] is a SIMD processing unit in the ARMv7 and ARMv8 architectures. It contains 32 64-bit registers. They can also be viewed as 16 128-bit registers. These registers are considered as vectors of elements of the same data type. The data type can be either 8-, 16-, 32-, or 64-bit signed or unsigned integer, or single-precision floating point. NEON provides an extension of ARM instruction set for vector operations, such as arithmetic, logical, and bitwise operations.

The basic way to use the NEON instructions is to write assembly code. However, this process is hard and tedious. It is also bug-prone. Another way is to use intrinsic functions provided by compilers, such as GCC and ARM RealView Compilation Tools. An intrinsic function call is replaced to a NEON instruction (or a sequence of instructions) by the compiler. Figure 3 shows an example of using NEON intrinsics. `vaddq_u32` adds two 128-bit wide vectors each containing four 32-bit unsigned integers. Using NEON intrinsics is much simpler than writing assembly code but still complicated. Programmers need to convert an expression into prefix notation and choose appropriate intrinsic functions for different vector widths and element types.

Our OpenCL C compiler optimizes vector operations and built-in functions in OpenCL C using the NEON intrinsics of GCC. It provides programmers more opportunity to vectorize their applications with much less programming effort. Moreover, GCC performs auto-vectorization on a work-group function and generates a binary containing NEON instructions, even if the kernel does not use vector operations explicitly.

### 3.1 Vector Operations

While our source-to-source translator generates a work-group function from a kernel, it also translates every vector operation into the form that can be vectorized by GCC. First, vector operations can be converted into one or more NEON intrinsic functions, as shown in Table 1. Since the built-in header defines the vector types as shown in Figure 4, the `neon` field is the vector itself having a data type for NEON intrinsic functions (e.g., `uint32x4_t` in Figure 3). A comparison operator (e.g., `==`, `<`, and `>`) is converted to two consecutive intrinsic function calls. The first function returns the result as a vector of unsigned integers, and the second function converts it into a vector of signed integers to satisfy the semantics of OpenCL C. A division operator (`/`) is implemented by the Newton-Raphson method. It first computes the reciprocal of $y$ and then multiplies it by $x$ to obtain $x/y$. All other kinds of operations are just replaced by a single intrinsic function call. The name of the intrinsic function is determined depending on the type of arguments. For example, `a + b` is converted to the `vadd_s32()` function if `a` and `b` are `int4` vectors, but it is converted to the `vaddq_s16()` function if `a` and `b` are `short8` vectors.

| Operation | NEON Intrinsics |
|---|---|
| `-a` | `vneg_s32(a.neon)` |
| `~a` | `vmvn_s32(a.neon)` |
| `a + b` | `vadd_s32(a.neon, b.neon)` |
| `a - b` | `vsub_s32(a.neon, b.neon)` |
| `a * b` | `vmul_s32(a.neon, b.neon)` |
| `x / y` | `float32x2_t t = vrecpe_f32(y.neon);`<br>`t = vmul_f32(vrecps_f32(y.neon, t), t);`<br>`t = vmul_f32(vrecps_f32(y.neon, t), t);`<br>`result = vmul_f32(x.neon, t);` |
| `a & b` | `vand_s32(a.neon, b.neon)` |
| `a \| b` | `vorr_s32(a.neon, b.neon)` |
| `a ^ b` | `veor_s32(a.neon, b.neon)` |
| `a == b` | `vreinterpret_s32_u32(vceq_s32(a.neon, b.neon))` |
| `a > b` | `vreinterpret_s32_u32(vcgt_s32(a.neon, b.neon))` |
| `a < b` | `vreinterpret_s32_u32(vclt_s32(a.neon, b.neon))` |
| `a << n` | `vshl_n_s32(a.neon, n)` |
| `a >> n` | `vshr_n_s32(a.neon, n)` |

**Table 1.** Conversion rules of vector operations. `a` and `b` is `int2` vectors, `x` and `y` is `float2` vectors, and `n` is an integer.

```
union __cl_uint4 {
  unsigned int v[4];
  uint32x4_t neon;
  ...
};
typedef union __cl_uint4 uint4;
```

**Figure 4.** The definition of `uint4` type in the built-in header.

Consider the kernel in Figure 5 (a). Figure 5 (b) shows the work-group function generated by our source-to-source translator. It contains a NEON intrinsic function call.

Alternatively, the source-to-source translator can convert a vector operation into an expression that performs the operation element-wisely. Figure 5 (c) shows an example for the kernel in Figure 5 (a). Some vector types (e.g.,`char4` and `double2`) and operations (e.g., remainder operations (`%`)) should be converted in this manner because NEON does not support them. On the contrary, if the vector operation is supported by NEON, the auto-vectorizer in GCC may convert the element-wise expression into NEON instructions. In this case, the binary generated from two conversion methods

```
__kernel void foo(__global int4* dst, __global int4* src) {
  int id = get_global_id(0);
  dst[id] = src[id] * src[id];
}
```

(a)

```
void foo(int4* dst, int4* src) {
  int id;
  for (__k = 0; __k < __local_size[2]; __k++) {
    for (__j = 0; __j < __local_size[1]; __j++) {
      for (__i = 0; __i < __local_size[0]; __i++) {
        dst[id].neon = vmulq_s32(src[id].neon, src[id].neon);
      }
    }
  }
}
```

(b)

```
void foo(int4* dst, int4* src) {
  int id;
  for (__k = 0; __k < __local_size[2]; __k++) {
    for (__j = 0; __j < __local_size[1]; __j++) {
      for (__i = 0; __i < __local_size[0]; __i++) {
        dst[id] = (int4){src.v[0]*src.v[0], src.v[1]*src.v[1],
                         src.v[2]*src.v[2], src.v[3]*src.v[3]};
      }
    }
  }
}
```

(c)

**Figure 5.** Sample code for vectorization; (a) a kernel; (b) the work-group function containing vector intrinsics for the kernel; (c) the work-group function containing vector operations for the kernel.

(i.e., using intrinsics and using element-wise operations) are not quite different. We address this issue in Section 5.3.

OpenCL enables a host program to get the preferred vector width of an OpenCL device using the `clGetDeviceInfo()` API function and to choose the best kernel implementation. Our OpenCL framework declares the preferred vector width to be 128 bits.

### 3.2 Built-in Functions

We use NEON intrinsics to optimize the following OpenCL C built-in functions.

***Type conversion functions.*** NEON supports instructions to set all elements in a vector to the same value (e.g., `vdup`), and to convert between a vector of 32-bit signed/unsigned integers and a vector of single-precision floating point values (e.g., `vcvt`). These instructions are used to optimize type conversion built-ins in OpenCL C.

***Math functions.*** Math functions can be vectorized in two different ways. Take the sine function as an example. First, we can vectorize the computational process of $\sin(x)$ to shorten the execution time of the function. Second, we can compute $\sin(x_1)$, $\sin(x_2)$, $\cdots$, $\sin(x_n)$ for different $x_1$, $x_2$, $\cdots$, $x_n$ at the same time using a vector operation. This does not affect the execution time of the sine function, but improves the throughput. It is beneficial for the vector versions of built-in math functions because they need to compute the result for every vector element. We adopt two open-source NEON-optimized math libraries as part of the OpenCL C built-in library. math-neon[1] provides various math functions vectorized in the first way. neon_mathfun[21] vectorizes `sin(x)`, `cos(x)`, `exp(x)`, and `log(x)` functions in the second way.

| Device | Device A | Device B | |
|---|---|---|---|
| Name | Nexus 7 | Snowball board v11 | |
| SoC | NVIDIA Tegra 3 T30L | ST-Ericsson Nova A9500 | |
| CPU | ARM Cortex-A9 | ARM Cortex-A9 | |
| # of cores | 4 | 2 | |
| Clock freq. | 1.2 GHz | 1.0 GHz | |
| L1I cache | 32 kB | 32 kB | |
| L1D cache | 32 kB | 32 kB | |
| L2 cache | 1 MB | 512 kB | |
| Memory | 1 GB (DDR3L-1333) | 1 GB (LPDDR2-800) | |
| OS | Ubuntu 13.04 | Ubuntu 12.04 | Android 2.3 |
| Compiler | gcc 4.7.2 | gcc 4.5.3 | PGCL 12.8 |

**Table 2.** Target devices.

***Image read/write functions.*** In GPUs, image objects can be accessed faster than buffer objects because they are typically stored in the texture memory and read/written through dedicated hardware. In CPUs, however, both buffer and image objects are stored in the main memory, and accessing image objects incurs additional overhead due to the built-in operations in the image read/write functions. Thus, image objects are rarely used in multicore CPUs. Nevertheless, we optimize the image read/write functions taking OpenCL applications written for GPUs into account. To do so, NEON instructions are used to normalize an image coordinate and to interpolate the result from multiple pixels in the image.

## 4. Experimental Setup

### 4.1 Target Devices

We use two target devices to evaluate our OpenCL framework. Device A (Nexus 7) comes with an NVIDIA Tegra 3 system-on-chip containing a Cortex-A9 quad-core processor and 1GB of main memory. Device B (Snowball board) comes with an ST-Ericsson Nova A9500 system-on-chip containing a Cortex-A9 dual-core processor and 1GB of main memory. Device B is used for the performance comparison between our OpenCL framework and the PGCL[26]. Our OpenCL framework runs on Ubuntu 12.04, while the PGCL runs on Android 2.3 Gingerbread. Device A is used for all other experiments. It runs Ubuntu 13.04. Table 2 describes the target devices in detail.

### 4.2 Benchmark Applications

We chose 37 benchmark applications from OpenCL samples in AMD APP SDK[4], Parboil benchmarks[25], and SNU NPB suite[22]. Some applications in the benchmark suites are excluded because they run too fast or require too large memory space.

Applications in AMD OpenCL samples only contain OpenCL versions (OCL), while applications in Parboil and SNU NPB provide sequential versions written in C (SEQ), OpenMP versions (OMP), and OpenCL versions. 10 applications in AMD OpenCL samples use one or more vector operations supported by NEON in the kernels. 9 applica-

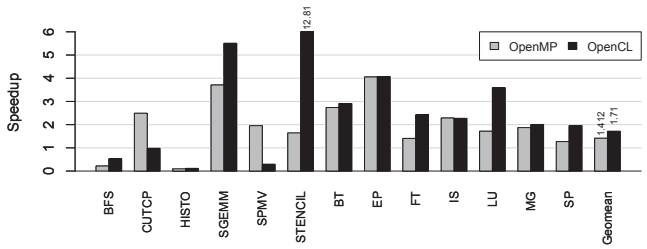


**Figure 6.** The speedup comparison between OMP and OCL versions. The speedup is obtained over SEQ versions.

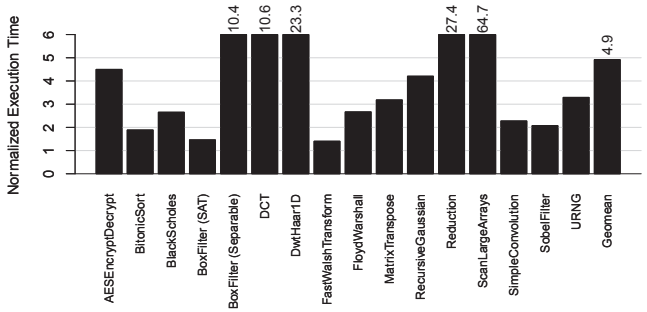


**Figure 7.** The execution time of applications executed with PGCL 12.8[26] normalized to that of our OpenCL framework. The higher the value is, the better our framework.

tions in AMD OpenCL samples use built-in functions that are optimized by NEON instructions (Section 3.2). Note that the vector operations or the built-in functions do not always dominate the performance of an application.

# 5. Evaluation

## 5.1 Comparison with OpenMP

We measure the speedups of OMP and OCL versions over SEQ versions to compare the effectiveness of two programming models, OpenMP and OpenCL. Figure 6 shows the result on Device A. Both OMP and OCL versions utilize all four cores in the ARM processor of Device A to execute computational kernels. OMP and OCL versions of BFS and HISTO invoke atomic operations frequently to share a queue (BFS) or histogram bins (HISTO) between threads. This is the reason whey they are slower than the SEQ versions. OCL versions of SGEMM and STENCIL have superlinear speedups because the array access patterns of the OCL versions differs from that of the SEQ versions. In most applications, OpenCL delivers performance comparable or higher than OpenMP. The average speedup of OCL versions is 1.71, while that of OMP versions is 1.41.

## 5.2 Comparison with PGCL

We compare the performance of our OpenCL framework with PGCL 12.8[26], an OpenCL framework that targets ST-Ericsson U8500 and later SoCs and Android platforms. Since PGCL only supports a subset of the OpenCL specification (i.e., the *embedded profile* of OpenCL) and has some

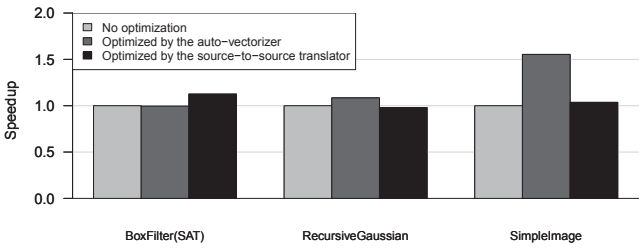


**Figure 8.** The speedups of applications obtained by the NEON optimizations for vector operations.

bugs, we cannot measure the performance of all applications on PGCL. Instead, we measure and compare the performance of the selected applications from AMD OpenCL samples. PGCL is executed with the `-fast` flag to enable the NEON optimizations of PGCL. Figure 7 shows the result on Device B. All applications run faster with our OpenCL framework. PGCL is 4.9 times slower than our OpenCL framework on average.

PGCL is known to use LLVM for generating machine code for the target device, so the quality of the ARM code generated by GCC in our OpenCL and LLVM in PGCL seems to make difference in performance. Because of PGCL's proprietary design, we cannot analyze technical details.

## 5.3 Effect of Optimizations for NEON

***Vector Operations.*** Figure 8 shows the speedups of three applications from AMD OpenCL samples, when vector operations in kernels are optimized with NEON instructions. We choose the applications that contain vector operations in the kernels and their performance is largely depends on the vector operations. Optimized by the auto-vectorizer shows the case that our source-to-source translator generates an expression that performs the operation element-wisely, and the auto-vectorizer in GCC actually converts it to NEON instructions. Optimized by the source-to-source translator shows the case that our source-to-source translator directly converts vector operations to NEON intrinsic function calls.

The result shows that the performance of BoxFilter is improved by the vectorization performed by our source-to-source translator, while that of RecursiveGaussian and SimpleImage is improved by the auto-vectorizer in GCC. There is a trade-off between the two vectorization methods. Our translator may exploit additional vectorization opportunities that the auto-vectorizer cannot find. On the contrary, the quality of the binary generated by the auto-vectorizer is better than that from the NEON intrinsic call, because GCC does not translate the intrinsic calls to the binary efficiently enough.

We test different versions of GCC from 4.5.3 to 4.7.2 in order to evaluate the effect of NEON optimizations in our OpenCL and notice that there is a significant improvement in the later version of GCC in terms of NEON code generation, which results in comparable performance level as our OpenCL. Particularly, the change in default vector size from 64 to 128 bit (in 4.7), the addition of support for wider in-

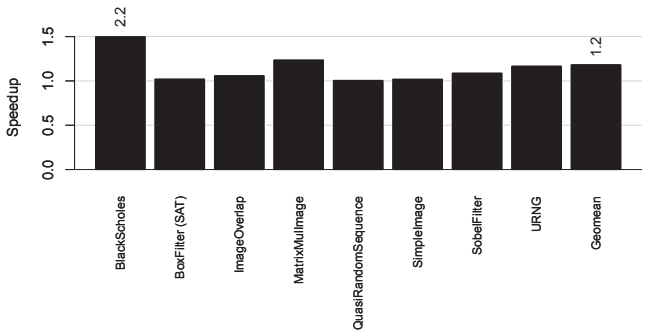


**Figure 9.** The speedups of applications obtained by the NEON optimization for built-in functions.

| Applications | # of auto-vectorized loops | # of loops in total |
|---|---|---|
| BFS | 0 | 4 |
| CUTCP | 2 | 10 |
| HISTO | 2 | 10 |
| SGEMM | 0 | 2 |
| SPMV | 0 | 1 |
| STENCIL | 0 | 2 |

**Table 3.** Loop-level auto-vectorizations performed by GCC.

| Before hand-vectorization | After hand-vectorization |
|---|---|
| 0.012682 seconds | 0.006404 seconds |

**Table 4.** The execution time of the OCL version of SGEMM before and after hand-vectorization.

structions and the revision of load/store multiples (in 4.6) are believed to contribute most to the performance gain in GCC's auto-vectorization.

***Built-in Functions.*** To show the impact of the NEON optimizations on built-in functions (Section 3.2), we choose the OCL versions of applications that use the optimized built-in functions, and measure the performance before and after the opimization. Figure 9 shows the result on Device A. Since the performance of BlackScholes largely depends on built-in math functions, we obtatin the speedup of 2.2. The performance of other applications also increases by up to 23.5%.

***Loop-level Auto-vectorization.*** The semantics of OpenCL allows to execute statements in different work-items in any execution ordering. Thus work-item coalescing loops (i.e., triply nested loops in Section 2.4) can be vectorized without problem. However, the auto-vectorizer in GCC cannot find this kinds of parallelism. Table 3 shows how many work-item coalescing loops in work-group functions are auto-vectorized by GCC. In four applications, GCC cannot find vectorizable loops at all. For CUTCP and HISTO, GCC finds two vectorizable loops for each application. However, these loops only contain a single statement and do not affect the overall performance.

Instead, programmers can vectorize kernels using OpenCL vector types explicitly. For an instance, we vectorize the OCL version of SGEMM in Parboil benchmarks by hand and measure the performance improvement on Device A. Table 4 shows the result. The performance of the hand-vectorized version is improved about twice.

## 6. Related Work

There are prior studies that propose methods to execute OpenCL or other SPMD threaded applications on multicore CPUs. Gummaraju *et al.*[10] introduce *Twin Peaks*, an OpenCL framework that targets heterogeneous processors containing both CPU and GPU cores. Lee *et al.*[18] propose an OpenCL framework for Cell BE processors. Their proposal can also be used for multicore CPUs. As introduced in Section 2.4, Twin Peaks use the `setjmp()` function to iterate over work-items in an NDRange, while Lee *et al.* use the work-item coalescing technique. Stratton *et al.*[23, 24] and Guo *et al.*[11] propose methods to translate a CUDA program into a parallel C program using a technique similar to the work-item coalescing. The state-of-the-art OpenCL frameworks[4, 12, 17] support multicore CPUs as their target devices.

Those studies mainly focus on desktop and server processors, such as x86 CPUs. However, we aim at an OpenCL framework for multicore ARM processors with the NEON vector unit. We show the effectiveness of OpenCL and our framework through the experiments on mobile devices. The PGCL[26] is the only OpenCL framework that has NEON support and is available in the market, which mainly targets ARM processors. The experimental result in Section 5.2 shows that our OpenCL framework outperforms the PGCL.

The SNU-SAMSUNG OpenCL framework[8], pocl[2], and COPRTHR[7] also supports ARM processors, but they do not have NEON support and optimizations. They just depends on auto-vectorization in a native compiler. Section 5.3 evaluates the effectiveness of auto-vectorization in GCC by focusing on work-group functions. As a result, we claim that auto-vectorization in GCC is not beneficial for the work-group functions and programmers need to vectorize kernels by hand to achieve high performance. Maleki *et al.*[19] evaluated auto-vectorization capabilities of three compilers (i.e., GCC, the Intel C compiler, and the IBM XLC compiler) using a variety of benchmarks to see the effectiveness of vectorizing compilers.

RenderScript[9] also provides an SPMD programming model and built-in vector operations for multicore ARM processors. However, it is a component of the Android operating system and does not support other kinds of platforms.

Karrenberg *et al.*[13, 14] propose an auto-vectorization technique for OpenCL kernels. Their technique finds parallelism at the level of work-item coalescing loops. Kerr *et al.*[15] propose a similar technique for CUDA kernels. We leave as our future work implementing auto-vectorization techniques in our framework and evaluating their performance on ARM processors.

## 7. Conclusions

In this paper, we introduce an OpenCL framework for multicore ARM processors. The ARM processor acts as both a

host processor and an OpenCL device. Every core becomes a compute unit and executes all work-items in the same work-group sequentially one by one. Work-groups in the NDRange are evenly distributed to the cores. Our OpenCL C compiler adopts the work-item coalescing technique to execute multiple work-items in a single work-group efficiently.

Programmers may utilizes the NEON vector unit in two ways to boost the performance of kernels. First, using OpenCL vector types and operations gives more vectorization opportunity to the kernel. The auto-vectorizer in GCC cannot find parallelism at the level of work-item coalescing loops. However, once a kernel is vectorized by hand, our source-to-source translator or the auto-vectorizer in GCC can easily convert the vector operations to NEON instructions. Second, our built-in functions of OpenCL C use NEON instructions in their implementation.

We evaluate the performance of our OpenCL framework using 37 benchmark applications from three benchmark suites and two multicore ARM SoCs. The result shows that our framework is 21% faster than OpenMP and 4.9 times faster than PGCL on average. We also present the effectiveness of the optimization techniques for NEON. As a result, we show that OpenCL is a promising programming model for multicore ARM processors.

## Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2013R1A3A2003664). ICT at Seoul National University provided research facilities for this study.

## References

[1] math-neon: ARM NEON optimised approximate cmath like library. `https://code.google.com/p/math-neon/`.

[2] pocl - Portable Computing Language. `http://pocl.sourceforge.net/`.

[3] Altera. OpenCL for Altera FPGAs: Accelerating performance and design productivity. `http://www.altera.com/products/software/opencl/opencl-index.html`.

[4] AMD. AMD accelerated parallel processing (APP) SDK. `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`.

[5] ARM. Mali OpenCL SDK. `http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/`.

[6] ARM. Introducing NEON Development Article. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/index.html`, 2009.

[7] Brown Deer Technology. COPRTHR (CO-PRocessing THReads). `http://www.browndeertechnology.com/coprthr.htm`.

[8] Center for Manycore Programming. SNU-SAMSUNG OpenCL framework. `http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU-SAMSUNG_OpenCL_Framework.html`.

[9] Google. RenderScript. `http://developer.android.com/guide/topics/renderscript`.

[10] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 205–216, New York, NY, USA, 2010. ACM.

[11] Z. Guo, E. Z. Zhang, and X. Shen. Correctly treating synchronizations in compiling fine-grained SPMD-threaded programs for CPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 310–319, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Intel. Intel SDK for OpenCL applications 2013. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`.

[13] R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society.

[14] R. Karrenberg and S. Hack. Improving performance of opencl on cpus. In *Proceedings of the 21st international conference on Compiler Construction*, pages 1–20, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] A. Kerr, G. Diamos, and S. Yalamanchili. Dynamic compilation of data-parallel kernels for vector processors. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 23–32, New York, NY, USA, 2012. ACM.

[16] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[17] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352, New York, NY, USA, 2012. ACM.

[18] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An OpenCL framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 193–204, New York, NY, USA, 2010. ACM.

[19] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[20] NVIDIA. OpenCL. `http://developer.nvidia.com/opencl`.

[21] J. Pommier. Simple arm neon optimized sin, cos, log and exp. `gruntthepeon.free.fr/ssemath/neon_mathfun.html`.

[22] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, pages 137–148, 2011.

[23] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Languages and compilers for parallel computing. chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[24] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, New York, NY, USA, 2010. ACM.

[25] The IMPACT Research Group. Parboil benchmarks. `http://impact.crhc.illinois.edu/parboil.aspx`.

[26] The Portland Group. PGI OpenCL compiler for ARM. `http://www.pgroup.com/products/pgcl.htm`.

[27] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa. *The OpenCL Programming Book*. Fixstars, Tokyo, Japan, 2010.