CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators

Geoffrey Ndu, Javier Navaridas and Mikel Luján
School of Computer Science
The University of Manchester Oxford Road
Manchester, M13 9PL, United Kingdom
firstname.lastname@manchester.ac.uk

ABSTRACT

Programming FPGAs with OpenCL-based high-level synthesis frameworks is gaining attention with a number of commercial and research frameworks announced. However, there are no benchmarks for evaluating these frameworks. To this end, we present CHO benchmark suite an extension of CHStone, a commonly used C-based high-level synthesis benchmark suite, for OpenCL. We characterise CHO at various levels and use it to investigate compiling non-trivial software to FPGAs. CHO is work in progress and more benchmarks will be added with time.

Categories and Subject Descriptors

A.1 [**General Literature**]: Introductory and Survey; B.6.3 [**Logic Design**]: [automatic synthesis]; C.1.3 [**Computer Systems Organization**]: Other Architecture Styles—*adaptable architectures*; D.1.3 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Measurement, Performance

Keywords

OpenCL, FPGA, High Level Synthesis, Accelerator

1. INTRODUCTION

Open Computing Language (OpenCL) [21] is an open standard for platform-independent, general purpose parallel programming across CPUs, GPUs and accelerators. OpenCL consists of an API for coordinating parallel computation and a cross platform programming language (a subset of ISO C99 with extensions for parallelism). It allows software to be written once and executed on any device that supports OpenCL. Its execution model consists of a host device which submits computational intensive kernels to compute devices for execution.

Programming Field-Programmable Gate Arrays (FPGAs) with OpenCL-based High-Level Synthesis (HLS) frameworks is now becoming mainstream with active support by the major FPGA vendors [27]. HLS is the automatic conversion of an algorithmic description into either a low-level Register Transfer Level (RTL) description or a digital circuit [10]. RTL refers to the low-level design abstraction that models a digital circuit in terms of the flow of digital signals between registers and the logical operations performed on those signals. HLS allows a designer to work more productively at a higher level of abstraction and achieve faster time-to-market than using error prone and difficult to debug RTL. Further, frameworks that use software programming languages, such as C and OpenCL, open up the power of FPGAs to software engineers (who outnumber hardware engineers by an order of magnitude). OpenCL has been used in implementing diverse algorithms on FPGAs [8, 11, 6].

Benchmarking is an important technique for analysing the performance of systems by studying the execution of the benchmark applications that are chosen to be a representation of the applications of interest. A good HLS benchmark suite should allow HLS framework developers to qualitatively evaluate new ideas as well serve as a standard for benchmarking the diverse HLS frameworks available. From our discussion with HLS users (especially non-FPGA experts), who have to choose from the myriad of HLS frameworks, the second objective is equally important as the first.

In this paper, we introduce CHO: a suite of benchmark applications for OpenCL-based HLS platforms that meets the objectives set out above. CHO is work in progress and presently is a rewrite of the C-based CHStone benchmark suite [17]. We will be adding more applications with time. CHStone is the commonly used HLS benchmark suite and consists of 12 applications from diverse application domains. Although based largely on C, OpenCL differs in some aspects from the standard C language. For example, OpenCL has disjoint memory spaces and moving data from one memory space to another need to be done explicitly. Hence, moving from one language to the other is often not straightforward.

This paper makes the following contributions:

- We present CHO an OpenCL port of the CHStone HLS benchmark enabling the benchmarking of OpenCL-based HLS.
- We characterise CHO at various levels.

- We use CHO and a state-of-the-art OpenCL HLS framework to evaluate compiling non-trivial programs to FPGA.

2. RELATED WORK

Benchmarking of HLS frameworks does not have a rich history when compared to benchmarking of software platforms and compliers. Early HLS framework developers tend to use their own choice of applications for evaluation. In the 90's, the HLS community attempted to standardize benchmarking by proposing the 1992 High-Level Synthesis Workshop Benchmarks [14] and the 1995 High-Level Synthesis Design Repository [30]. These benchmarks covered a number different applications and application domains but were mostly written in algorithmic VHDL. VHDL is a type of Hardware Description Language (HDL) (specialized language for encoding the structure, design and operation of electronic circuits). However, HLS frameworks has since moved from HDLs to high-level software languages, mostly variants of C. These benchmark suites have a few C-based applications but they are mostly tiny (less than a 100 lines of code) Digital Signal Processing (DSP) kernel loops [17]. Consequently, these benchmarks are rarely used nowadays.

CHStone [17] is now the de-facto standard benchmark suite used in the HLS community but it lacks support for OpenCL. There are OpenCL benchmark suites used in evaluating heterogeneous computing platforms such as Valar [26], Rodina [7] and SHOC [12] but they are too large and complex for FPGA synthesis and fitting.

Commercial and research OpenCL frameworks are increasingly being developed [27]. Notable frameworks include OpenCL-to-Silicon framework [29], SDAccel Development Environment [32] and Altera SDK for OpenCL (AOCL) [5] (the first HLS to pass OpenCL conformance tests [5]). To the best of our knowledge none of the OpenCL frameworks were benchmarked with any standard benchmark suite.

3. BACKGROUND

3.1 FPGAs

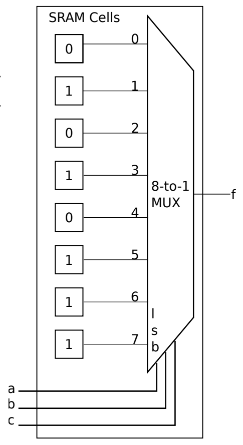
An FPGA is essentially a sea of Look-Up Tables (LUTs). A LUT is a small high-speed memory and is programmed by loading a function's truth table as shown in Figure 1. Combining a LUT and a D flip-flop (a circuit with two stable states and can be used to store information) results in what is often referred to as a logic cell. Several logic cells together with special-purpose circuitry such as an adder/subtractor carry chain form a logic block. Logic blocks can be connected to other logic blocks through a reconfigurable routing network making it possible to implement complex functions. FPGAs also contain other components such Block RAMs (BRAMs), DSP slices, various communication interfaces (e.g. PCI Express) and even processor cores. Figure 2 shows a typical high-end FPGA.

Modern FPGAs can have up to 4 million logic cells and 89 Mbits of BRAMs. Typically, FPGAs run at a much lower clock frequency than CPUs and GPUs but can outperform them by implementing custom and often more power efficient execution pipeline [22, 31].

Figure 3 shows how a slower FPGA can outperform a CPU. The figure compares the implementation of 32-bit integer bit reversal [13] on an ARM 9 processor that requires about 38 instructions and 48 cycles, to an FPGA where the same operation can be performed by simply reversing the connections between two buffers. It is assumed that CPU and the FPGA implementation operate at the same frequency.

a	b	c	ab + c
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table



(b) LUT implementation

Figure 1: Implementing logic function $f(a, b, c) = ab + c$

Despite the advantages offered by FPGAs their use is still largely restricted to a narrow segment of hardware programmers as programming often involves writing complex RTL code. HLS raises the level of abstraction (and productivity) by allowing algorithmic descriptions to be converted into RTL or even a digital circuit. Most HLS frameworks are now C-based and often support only a subset of the parent language.

3.2 OpenCL Architecture

An OpenCL computing platform consists of a CPU host connected to one or more OpenCL devices as shown in Figure 4. The part of an application targeting the devices is called the kernel. A kernel is hardware agnostic and should run on any device that supports the OpenCL standard. Kernels are written in OpenCL C programming language, a variant/subset of C-99.

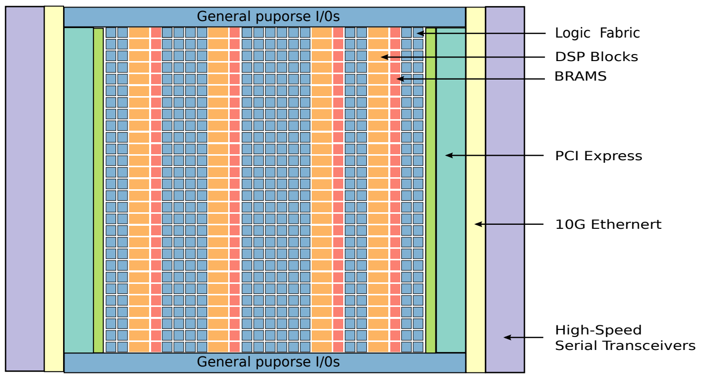
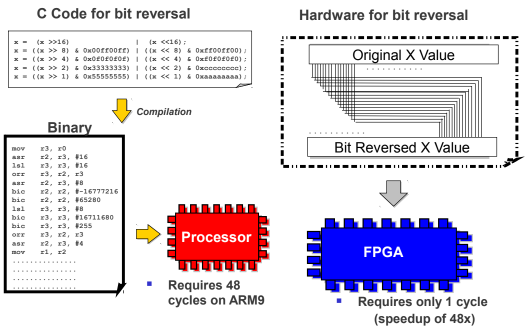


Figure 2: FPGA architecture


C Code for bit reversal
Compilation
Binary
Processor
Requires 48 cycles on ARM9
Hardware for bit reversal
Original X Value
Bit Reversed X Value
FPGA
Requires only 1 cycle (speedup of 48x)

```
x =  (x >>16)                | (x <<16);
x = ((x >> 8) & 0x00ff00ff) | ((x << 8) & 0xff00ff00);
x = ((x >> 4) & 0x0f0f0f0f) | ((x << 4) & 0xf0f0f0f0);
x = ((x >> 2) & 0x33333333) | ((x << 2) & 0xcccccccc);
x = ((x >> 1) & 0x55555555) | ((x << 1) & 0xaaaaaaaa);
```

```
mov   r3, r0
asr   r2, r3, #16
lsl   r3, r3, #16
orr   r3, r2, r3
asr   r2, r3, #8
bic   r2, r2, #-16777216
bic   r2, r2, #65280
lsl   r3, r3, #8
bic   r3, r3, #16711680
bic   r3, r3, #255
orr   r3, r2, r3
asr   r2, r3, #4
mov   r1, r2
...............
...............
...............
...............
```

(a) ARM9 bit reversal (b) FPGA bit reversal

Figure 3: How FPGAs outperform CPUs and GPUs

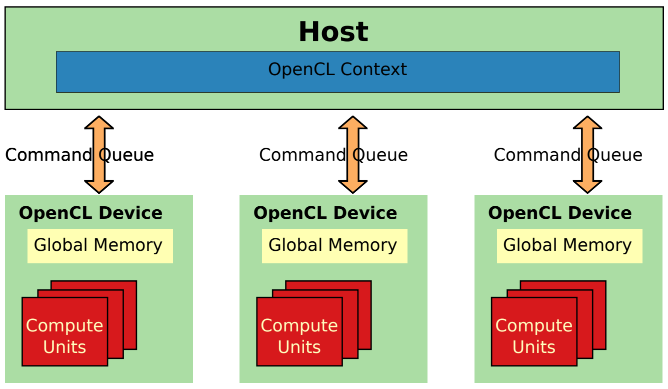


Figure 4: OpenCL Platform Architecture

The other part of an OpenCL application, the host, executes on the CPU and submits commands to perform the computations expressed in the kernels on devices. A kernel instance is called a work-item and multiple instances can be grouped into independent work-groups.

Two types of programming models are explicitly supported within OpenCL; the data parallel programming model and the task parallel programming model. In the task parallel programming model, a kernel is executed using a single work-item within a work-group while in the data parallel programming model multiple work-items are employed with input data partitioned across work-items.

The host defines a context for the execution of the kernels. The context includes the following resources: devices (devices to be used by the host), kernels (functions to run on devices), program objects (program sources and executables that implement the kernels) and memory objects(memory objects visible to the host and devices which contain values that can be operated on by the kernel instances). The context is created and manipulated by the host via OpenCL API.

Work-items have access to 4 disjoint memory regions: global memory (read/write memory accessible by all work-items), constant memory (read-only memory accessible to all work-items), local memory (read/write memory local to a work-group) and private memory (read/write memory private to a work-item)

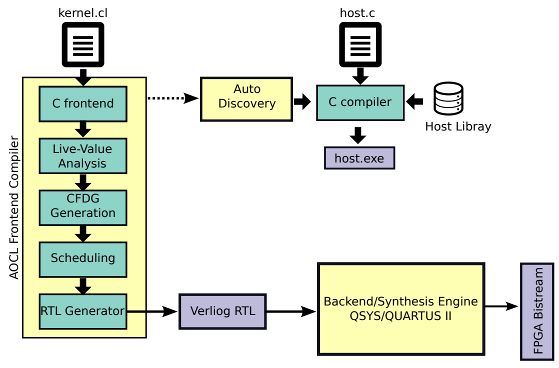


Figure 5: Altera OpenCL SDK (AOCL)

3.3 OpenCL HLS Compilers

A typical OpenCL HLS framework can be divided into two parts, the front-end and the back-end. The front-end converts the kernels to RTL code while the back-end compiles the RTL down to FPGA bitstream (the configuration data that is loaded into the FPGA).

Figure 5 shows the structure of AOCL [11], the first commercial OpenCL-based HLS framework. The front-end leverages LLVM compiler infrastructure [23]. LLVM is increasingly used as the front-end of C-based HLS frameworks. The C-language front-end parses the kernel into LLVM Intermediate Representation (IR). LLVM IR uses simple RISC-like instructions augmented with high-level information such as types and explicit control flow graphs to represent programs (see Figure 6 and Figure 7). Notice that the LLVM instructions in Figure 7 are simple enough to translate directly into hardware operations (e.g., a load from memory, an arithmetic computation). Hence, the rest of the front-end operates directly on the IR [11]. Each basic block is analysed and a control data flow graph is created for operations within a block. Each basic block is a hardware module that takes inputs from either the kernel arguments or another basic block, processes the data according to the instructions within and then produces output that is passed to other basic blocks. The basic-blocks are connected together to produce a complete circuit.

The scheduler attempts to minimize execution time as well as area. Finally, the RTL generator converts the IR instructions into Verlog RTL. The compiler automatically generates interfaces for accessing the off-chip memory (which serves as OpenCL's global memory) and the PCI Express link (used by the host program to access the global memory). The traditional hardware compilation (synthesis) tools are used to synthesize the RTL code for FPGA.

The host program can be compiled with any C/C++ compiler and linked against AOCL's host library. This library implements the OpenCL function calls that launch the kernels on an FPGA. A module embedded inside the FPGA, termed the Auto-Discovery module, allows the host program to query the FPGA about the kernels it holds. AOCL performs offline compilation only.

AOCL creates pipelined circuits to increase performance and maps work-items to pipeline stages. Let us assume that

```
__kernel void vector_add(__global int *a, __global ↩
    int *b, __global int *c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[1];
}
```

Figure 6: Vector addition in OpenCL

```
entry:
  %call = tail call i32 (i32, ...)* bitcast (i32 (↩
      ...)* @get_global_id to i32 (i32, ...)*)(i32 ↩
      0) #2
  %idxprom = sext i32 %call to i64
  %arrayidx = getelementptr inbounds i32* %a, i64 ↩
      %idxprom
  %0 = load i32* %arrayidx, align 4, !tbaa !2
  %arrayidx1 = getelementptr inbounds i32* %b, i64 ↩
      1
  %1 = load i32* %arrayidx1, align 4, !tbaa !2
  %add = add nsw i32 %1, %0
  %arrayidx3 = getelementptr inbounds i32* %c, i64 ↩
      %idxprom
  store i32 %add, i32* %arrayidx3, align 4, !tbaa ↩
      !2
  ret void
```

Figure 7: Vector addition in LLVM IR

AOCL has created a three stage (load, add and store) pipelined hardware for the kernel in Figure 6, as shown in Figure 8. On the first clock cycle, work-item 0 is clocked into the load stage. The circuit starts the processing of work-item 0 with fetching the first element of data from arrays a and b. On the second cycle, work-item 1 is clocked in while work-item 0, which has completed its read from memory and stored the results in the registers, moves to the second stage of the pipeline. Processing completes when the last work-item (work-item 7) exits from the last stage of the pipeline. For work-groups with a single work-items (i.e. kernels written in a task parallel fashion), AOCL would attempt to pipeline loops within kernels, using loop pipelining [4], to improve performance.

4. CHO BENCHMARK SUITE

4.1 Overview

CHO extends the popular C-based CHStone HLS benchmark suite providing a means of benchmarking present and emerging OpenCL frameworks. The kernels in CHO are implemented using OpenCL task parallel model (i.e. one work-item per work-group). This requires minimal changes to the overall structure of the original code. Further, some of the programs are difficult to express using the data parallel programming model.

CHO, as its progenitor, consists of 12 diverse applications as shown in Table 1. CHO targets OpenCL 1.0, the minimum standard, allowing it to be compiled by any OpenCL compliant compiler as OpenCL is designed to be backward compatible. Like CHStone we do not prescribe a particular way of using CHO.

Porting CHStone to OpenCl required eliminating the use of non-constant global variables as global variables in OpenCL must be constants. We made extensive use of structures

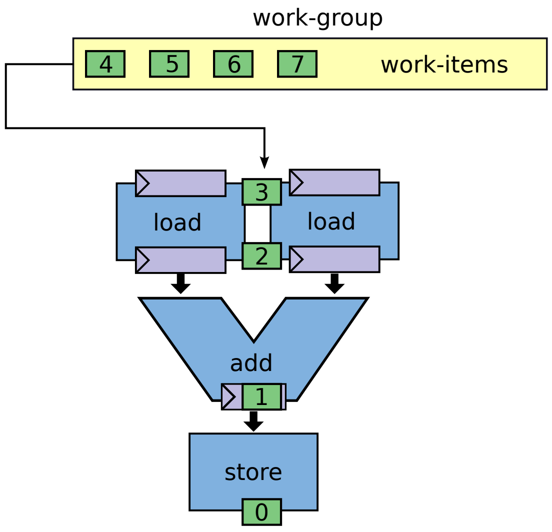


Figure 8: AOCL pipeline implementation

(`struct`) to pass state from one function to another. To improve performance we allocated variables whenever possible in the private memory space as it is by definition the fastest memory in OpenCL. To allow compilers to perform memory optimization we added `restrict` keyword to pointer arguments whenever possible. The `restrict` informs compilers that pointers that share the same type do not alias.

Each application in CHO is a single kernel and has a test-bench that is part of the host program. Each kernel reads/writes its input/output data from/to the host side (via PCIe and the global memory). Implementing the test-bench on the host instead of encoding in the kernel enables the study of how an OpenCL HLS compiler would handle external I/O interfacing between the FPGA and the host (assuming that is done automatically by the framework). I/O bandwidth and latency and its interfacing are critical components in FPGA design. Often implementations need to be specialized based on achievable external input and output bandwidth. We reused the test-vectors from CHStone.

4.2 CHO Kernels

The kernels in CHO are diverse, substantial, real world applications. They have all been shown to map well to FPGA using different languages, tools and technique making them ideal for benchmarking HLS frameworks targeted at FPGAs. Consequently, the suite is biased towards “embedded computing” where FPGAs have been used for decades for implementing algorithms. We describe briefly, next, the applications in the suite.

dfadd: is an implementation of IEC/IEEE-standard double-precision floating-point addition using 64-bit integers.

dfdiv: is an implementation of IEC/IEEE-standard double-precision floating-point division using 64-bit integers.

dfmul: is an implementation of IEC/IEEE-standard double-precision floating-point multiplication using 64-bit integers.

Domain	Application	Description	Original Sources
Arithmetic	dfadd	IEC/IEEE double-precession floating-point addition	SoftFloat [18]
	dfdiv	IEC/IEEE double-precession floating-point division	SoftFloat [18]
	dfmul	IEC/IEEE double-precession floating-point multiplication	SoftFloat [18]
	dfsin	Double-precession floating-point sine function	SoftFloat [18]
Media	adpcm	Adaptive differential pulse code encoder & decoder	SRTB [25]
	gsm	GSM residual pulse excitation/long term prediction coding	MediaBench [9]
	jpeg	JPEG image decoder	CHStone [17], PVRG [20]
	motion	Motion vector decoding for MPEG-2	AiLab [1]
Cryptography	aes	Implementation of Advanced Encryption Standard	AiLab [1]
	blowfish	Blowfish Encryption Algorithm	MiBench [16]
	sha	Secure Hash Algorithm	MiBench [16]
Miscellaneous	MIPS	Simplified MIPS processors	CHO [17], SoftFloat [18]

Table 1: CHO Kernels

dfsin: implements the sine function using 64-bit integers. It has several functions in common with the previously listed kernels.

adpcm: is an implementation of ITU G.722 Adaptive Differential Pulse-Code Modulation (ADPCM) algorithm used in the encoding and decoding audio signals. It is often used in Voice over IP communications. The kernel includes encoding and decoding functionalities.

gsm: is an implementation of Linear Predictive Coding, a method of encoding good quality speech at a low bit rate.The kernel implements only the lossy sound compression used in GSM (a mobile communication protocol)

jpeg: is an implementation of the JPEG still picture compression standards.

aes: implements the AES symmetric-key (the same key is used for encrypting and decrypting data) cyrpto-system. Encryption and decryption modules are implemented.

blowfish: is an implementation of the encryption function of Blowfish. Blowfish is a symmetric-key block cipher.

sha: implements the Secure Hash Algorithm, a cryptographic hash function.

4.3 Functional Verification

We tested CHO to ensure that we correctly implemented each kernel. First, we verified that the kernels can be parsed by any standard-complaint OpenCL 1.0 front-end. Since OpenCL-based HLS frameworks often rely on LLVM for their font-end processing we used Clang's [2] OpenCL 1.0 parser for our tests. Clang is a compiler front-end that converts C-based programming languages into LLVM IR. Syntax checking of kernels is important as OpenCL compilers are often varyingly lax in enforcing syntax correctness.

We validated that the kernels are functionally correct and produce the right results by compiling and running all kernels on different OpenCL devices. The devices used are shown in Table 2. All the kernels compiled and produced the correct results on execution.

Device	Type	Driver/Compiler
Intel Core i5	CPU	Intel OpenCL SDK 2013R3
Intel Core i5	CPU	AMD APP SDK 2.9
Intel Xeon E3	CPU	Intel OpenCL SDK 2013R3
Intel Xeon E3	CPU	AMD APP SDK 2.9

Table 2: Functional Test Platforms

5. CHARACTERIZING CHO

We characterised CHO at the source-level, after conversion into Abstract Syntax Tree (AST) and the IR-level, after conversion to LLVM-IR.

5.1 Source-Level Characterization

We implemented a Clang plugin that walked each kernel's AST and classified every token. We run the plugin on the AST produced by the Clang compiler. Table 3 is a summary of the source-level characteristics of the each kernel. 'Dominant Type' in the table refers the representative data type in each kernel while 'LoC' refers to the lines of code in the source excluding comments and empty lines. The table shows that kernels are non-trivial implementations with hundreds of lines of code, multiple functions plus diverse statements and operators.

Some kernels perform multiplications and divisions which are expensive. On FPGAs, division is the most expensive operator to implement followed by multiplication [24]. Modern FPGAs often have DSP blocks with high-speed embedded multipliers making it possible to multiply without using logic resources. However, there are only a few of these on FPGAs which implies that kernels that perform a lot of multiplications may still have to implement some of the multipliers

Kernel	Dominant Type	LoC	Functions	Variables		Statements					Operators						
				Scalar	Aggregate	*for*	*if*	*goto/break*	*switch*	*while*	Divide	Multiply	Add/Sub	Compare	Shift	Assign	Logic
adpcm	32-bit *int*	463	21	103	13	10	17	1	0	0	4	36	112	27	29	171	7
aes	32-bit *int*	667	10	42	25	16	24	31	6	0	14	95	98	41	184	389	209
blowfish	8-bit *char*	495	4	38	16	6	9	0	0	4	0	9	178	16	123	267	267
dfadd	64-bit *int*	352	81	2	18	1	43	6	0	0	0	0	29	46	21	104	34
dfdiv	64-bit *int*	280	19	96	1	1	31	0	0	2	2	4	33	45	30	100	31
dfmul	64-bit *int*	350	16	74	1	1	28	0	0	0	0	4	25	38	24	90	31
dfsin	64-bit *int*	611	31	178	1	1	74	6	0	3	2	7	54	91	45	188	65
gsm	32-bit *int*	310	12	45	7	17	16	0	0	1	0	47	168	57	23	211	11
jpeg	32-bit *int*	1061	31	204	30	34	58	11	1	11	3	52	135	92	44	342	21
mips	32-bit *int*	215	1	17	3	2	3	35	3	4	0	2	12	10	22	57	23
motion	32-bit *int*	436	13	59	15	7	21	0	0	4	2	6	40	28	16	84	7
sha	64-bit *char*	178	8	42	15	10	2	0	0	4	2	1	42	16	23	90	32

Table 3: Source-Level Characteristics

Optimization & Transformation	Description
inline	Bottom-up inlining of functions into callees
jump-threading	Attempts to find distinct threads of control flow running through a basic block
simplifycfg	Removes dead code and merges basic-blocks
loop-rotate	Simple loop rotation
gvn	Removes fully and partially redundant instructions plus redundant load elimination
instcombine	Combines instructions into fewer and simpler instructions

Table 4: LLVM transformations and optimizations

Kernel	Number Basic Blocks	Number Instructions	Number Loops
adpcm	95	2240	19
aes	186	5685	23
blowfish	58	5441	10
dfadd	199	1421	1
dfdiv	115	1212	3
dfmul	90	846	1
dfsin	469	4216	2
gsm	149	1493	18
jpeg	661	10261	109
mips	46	472	3
motion	1743	15121	326
sha	79	1364	30

Table 5: IR-Level Characteristics

using logic resources. All divisions must be implemented with logic resources which may use-up logic resources if there are many of them. Generally, these expensive functions can be shared reducing the number required as well as performance.

5.2 IR-Level Characterization

Since LLVM IR instructions are simple enough to directly correspond to hardware operations characterizing at the IR-level provides a more accurate and detailed picture about how kernels may be mapped to FPGAs. For instance, a single loop at the source-level may be simplified and split into multiple loops at the IR-level.

For IR-level characterization, we implemented a Clang/LLVM compiler that mimics the front-end of a generic OpenCL HLS compiler. Our custom compiler first translates OpenCL into LLVM IR and then applies transformations and optimizations [19] that has been shown to improve performance on FPGAs. These optimizations are shown in Table 4.

Table 5 summarizes the IR-level characteristics of CHO kernels. Notice that the number of loops has increased moving from source-level to IR-level as a result of running `simplify-cfg`. The compiler splits complex loops into multiple simpler efficient loops. Figure 9 is a breakdown of the IR-level instructions by instruction types. In the diagram, 'Memory' refers to load and store type instructions plus address generating instructions while 'Control' refers to instructions, such as branch, that transfer control from one part of a kernel to another. For some kernels e.g. **gsm** the expensive operators, multiplication and division, are still significant.

6. SYNTHESIZING CHO

In this section, we present the results of our attempts at synthesizing kernels in the CHO benchmark suite using AOCL, a state-of-the-art OpenCL HLS framework. The main objective is to determine if we can synthesize unmodified OpenCL kernel for FPGA. We have not modified the kernels so they are the same as those that run on the 'software' platforms in Table 2.

6.1 Target Compiler and Platform

We use AOCL 14.1.1.190 and target Nallatech P385-A7 FPGA accelerator card. The P385-A7 (its features are described in table Table 6) is based on Altera's Stratix V GX-A7 FPGA.

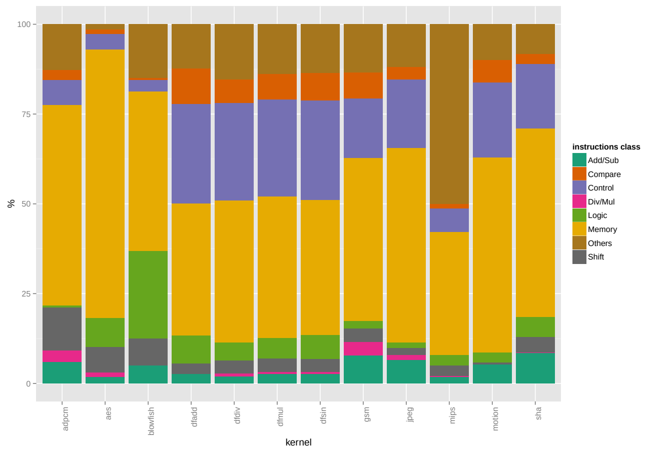


Figure 9: LLVM IR instruction classes

FPGA	Altera 5SGXMA7H2F35C2
Form Factor	Half-length Half-height PCIe Card
Host Interface	8-lane Gen3 PCIe
Memory	8GB DDR3 SDRAM

Table 6: P385-A7 FPGA accelerator card

AOCL supports augmenting OpenCL kernels by specifying kernels attributes. The available attributes are listed below.

- `packed`: Specifies that AOCL should not enforce data alignment requirements.
- `aligned(N)`: Used to specify the amount of data structure padding to be added to a data member. `N` is the amount of padding
- `reqd_work_group_size`: Specifies the required work-group size allowing AOCL to allocate the exact amount of FPGA resources needed to manage work-items in a work-group.
- `max_work_group_size`: Defines the maximum number of work-items that can be allocated within a work-group.
- `num_compute_units`: Sets the number of compute units to be instantiated to process the kernel. AOCL distributes the work-groups within a kernel across the specified number of compute units increasing throughput. This may increase global memory bandwidth contention among compute units.
- `num_simd_work_items:` This is similar to the attribute above but involves only replicating the datapath in a single instruction multiple data (SIMD) fashion. For AOCL to implement a SIMD datapath, the value of `num_simd_work_items` must evenly divides the value specified for `reqd_work_group_size`.
- `local_mem_size`: Defines the size of local memory allocated to pointer argument with local specifier. The default is 16 kB.

Adaptive Logic Modules (K)	235
Registers (K)	939
M20K Memory Blocks	2,560
M20K Memory (MBits)	50
Variable Precision Multipliers (18x18)	512
Variable Precision Multipliers (27x27)	256
DSP Blocks	256
Fabric clock (MHz)	800

Table 7: Starix V GX-A7 features

AOCL also supports the `#pragma unroll <N>` directive. This directive informs AOCL to attempt to unroll a loop `N` times. Omitting `N` directs AOCL to try complete unrolling.

Table 7 shows the features of Altera Stratix V GX-A7. In the table, `Adaptive Logic Modules` refers to Altera's logic blocks. Each block has 8 inputs with a Adaptive Look-Up Table (ALUT) (Altera's name for LUTs), 2 adders and 4 registers. `Variable Precision Multipliers` are the embedded, high-speed, variable precession multipliers. They can perform 9-bit, 18-bit, 27-bit and 36-bit word lengths operations. These multipliers are part of a larger structure called a DSP block. Stratix V devices contain dedicated memory blocks (Block RAMs), each 20-Kb in size, called M20K blocks. The number of blocks and the total number of RAM bits available from all M20K blocks are given in the table.

6.2 Sythesis Results

We were unable to synthesize and run every kernel in CHO for various reasons. However, we have seen a two-fold increase in the number of synthesizable kernels compared to the previous major version of AOCL [28]. For **jpeg** and **motion** the front-end stopped after internal compiler error. A summary of our attempts at synthesizing the kernels is presented as Table 8. In the table, ✓ means that a kernel was successfully synthesised. After synthesis, **adpcm**, **mips** and **sha** failed to produce the correct results while **blowfish** and **gsm** failed to run to completion. We were able to synthesize **blowfish** and run it using an older version (13.1) of AOCL.

The kernels were compiled using the attributes shown in Table 9. In addition to the attributes in the table, we used `#pragma unroll (N)` directive to unroll loops with statically determinable loop bounds. Table 10 presents the synthesis results.

Note that **blowfish** in Table 9 was compiled with AOCL 13.1 as explained earlier while the rest of the kernels was compiled with AOCL 14.1, hence the different set of attributes used. In AOCL 13.1, `max_share_resources` limits the number of times an operator e.g. multiplier can be reused without reducing computational throughput, `num_share_resources` sets the number of times an operator can be reused and `max_unroll_loops` limits the number of times AOCL can unroll each loop in a kernel. The attributes used for **blowfish** were set automatically by AOCL.

In Table 10, 'Loops' refers to the number of loops at the

Kernel	Synthesizable	Notes
adpcm	✓	
aes	✓	
blowfish	✓	
dfadd	✓	
dfdiv	✓	
dfmul	✓	
dfsin	✓	
gsm	✓	
jpeg	✗	Front-end error
mips	✓	
motion	✗	Front-end error
sha	✓	

Table 8: Summary of Synthesis

Kernel	Attributes Settings
aes	`num_compute_units(1)` `reqd_work_group_size(1,1,1)`
blowfish	`max_unroll_loops(1)` `num_compute_units(2)` `num_share_resources(1)` `max_share_resources(8)`
dfadd	`num_compute_units(1)` `reqd_work_group_size(1,1,1)`
dfdiv	`num_compute_units(1)` `reqd_work_group_size(1,1,1)`
dfmul	`num_compute_units(1)` `reqd_work_group_size(1,1,1)`
dfsin	`num_compute_units(1)` `reqd_work_group_size(1,1,1)`

Table 9: OpenCL attributes

LLVM-IR level while 'Pipelined Loops' refers to the number of those loops that AOCL managed to fully pipeline i.e. loops without Loop-Carried Dependencies (LCD) [4]. AOCL attempts to pipeline [15] (overlapping computations for different loop iterations in time and space) all loops so as to have a fully pipelined hardware structure. Some of the loops may have LCD which requires AOCL to generate extra hardware to account for these dependencies reducing throughput. Therefore, rewriting loops (following Altera's recommendations [3]) to minimize LCD is a good approach to improving performance. Pipelining in the presence of LCD is an active research area and techniques developed in this area could be beneficial to future HLS frameworks.

LCD is particularly a problem for **blowfish** where 3 of its loops each causes, on the average, a 567 cycles pipeline stall (i.e. successive iterations launched every 567 cycles) The cumulative effect of this is a significant degradation of performance as shown in the next section. Note that information about loops are available from the AOCL during compilation.

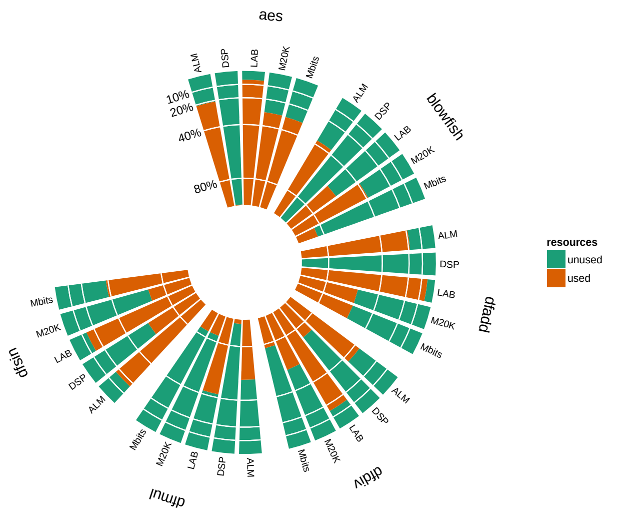


Figure 10: Percentage resource utilization

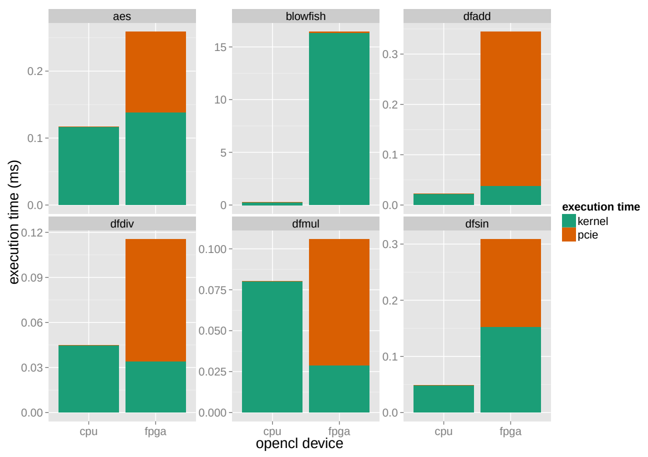


Figure 11: Comparing performance

Notice that the number of loops in Table 5 and Table 10 don't exactly correspond. This is because our custom compiler and AOCL may not be using the same set of optimizations and versions. Further, AOCL completely unrolls some of the loops as could be inferred from **dfmul**.

Figure 10 shows the percentage resource usage in terms on ALMs, LABs, DSP blocks, M20K blocks, total number of memory bits (Mbits in the figure). It gives a rough indication of the "complexity" of each implementation on the FPGA.

6.3 Performance

Figure 11 compares the execution time of the successfully synthesized kernels on the FPGA platform and on a CPU platform. The details of the CPU platform is shown as Table 11. Note that OpenCL is not performance portable and that we didn't manually optimize the kernels for FPGA. Compilation for the CPU was done offline to match AOCL. AOCL presently doesn't support out-of-order execution i.e. the execution of multiple tasks in parallel therefore performance measurements was done with a single task executing on the CPU and the FPGA.

Kernel	fmax (MHz)	ALMs	LABs	ALUTs	Registers	M20K	Memory Bits	DSP Blocks	Synthesis Time (Hours:Mins)	Loops	Pipelined Loops
aes	150	185 751	21 922	210 557	297 648	1788	36 618 240	1	05:50	15	11
blowfish	200	147 981	19 724	151 907	270 884	1567	7 946 172	0	03:00	11	7
dfadd	220	190 067	22 142	190 489	267 431	1131	23 162.880	0	01:15	1	1
dfmul	223	104 976	13 479	112 579	163 412	389	2 050 512	160	01:43	1	1
dfdiv	117	117 403	19 929	110 387	275 825	1111	4 096 652	72	03:45	1	0
dfsin	107	195 113	12 914	98 391	368 809	1576	32 276 480	162	02:59	10	6

Table 10: Synthesis results

CPU	Intel Xeon CPU E31245 @ 3.30GHz
Memory Size	16 GB DDR3
OpenCL SDK	Intel Kernel Builder for OpenCL 1.4.0.117

Table 11: CPU platform features

FPGA outperforms the CPU, without the overhead of PCIe data transfer, except for **aes**, **blowfish** and **dfsin**. For **blowfish**, the FPGA is slower than the CPU, by an order of magnitude. As mentioned earlier, **blowfish** suffers from having loops that are significantly affected by LCD.

7. CONCLUSION

In this paper, we presented CHO, presently an OpenCL port of the commonly used CHStone HLS benchmark. CHO aims to enable HLS framework developers to qualitatively evaluate new ideas as well as enable the benchmarking of the increasing number of OpenCL HLS frameworks.

We characterised CHO at the OpenCL source-level and at the LLVM IR-level. We showed that kernels in the benchmark suite are substantial. IR-level characterization provides more detailed information as the LLVM IR instructions are simple enough to directly correspond to hardware operations.

We synthesized all but 2 of the kernels. However, only 6 of the 12 kernels ran to completion and produced the correct results. We recognize that AOCL is relatively new and could improve with time. In fact, we have noticed that the number of kernels that are synthesizable and runnable seems to increase with each new version of the compiler.

We showed that it is straightforward to compile unmodified OpenCL kernels down to FPGA (when compilation works). However, simply compiling algorithms designed for GPUs and CPUs to FPGAs may not lead to performance improvements.

There are a number of possible directions for future work.

- The programs in CHStone, which we ported to OpenCL, are skewed towards ”embedded computing” where FPGAs has been used for decades. We are looking at introducing programs from other areas, such databases, that have been shown to have efficient FPGA implementations.
- CHO employs, exclusively, OpenCL's task parallel programming model. However, an OpenCL-based HLS framework may compile the two programming models differently as is the case with AOCL. We are looking at introducing kernels into CHO that can be mapped efficiently onto the data parallel programming model of OpenCL.

8. SOFTWARE DOWNLOAD

CHO is publicly available at `http://it302.github.io/cho`.

Acknowledgement

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 318633. We wish to thank Altera University Program for their software donation. Mikel Luján is supported by a Royal Society University Research Fellowship.

9. REFERENCES

[1] AILab. `http://www-ailab.elcom.nitech.ac.jp`.

[2] Clang. `http://clang.llvm.org`.

[3] Altera SDK for OpenCL: Optimization Guide. Dec. 2013.

[4] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, Sept. 1995.

[5] Altera Corporation. *Altera SDK for OpenCL: Programming Guide*, ocl002-13.1.1 edition, Dec. 2013.

[6] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[7] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Dec 2010.

[8] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 297–304, Jan 2013.

[9] L. Chunho, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, Dec 1997.

[10] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *Design Test of Computers, IEEE*, 26(4):8–17, July 2009.

[11] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on FPGAS. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012.

[12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, 2010.

[13] H. G. Dietz. The Aggregate Magic Algorithms. Technical report, University of Kentucky.

[14] N. Dutt. Benchmarks for the 1992 High Level Synthesis Workshop. 1992.

[15] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. 2009.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

[17] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[18] J. R. Hauser. SoftFloat Release 2b General Documentation. 2007.

[19] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 89–96, April 2013.

[20] A. C. Hung. PVRG-JPEG CODEC 1.1. Technical report, Stanford University, 1993.

[21] Khronos Group Inc. The OpenCL Specification Version: 1.0, June 2009.

[22] D. Koch and J. Torresen. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 45–54, 2011.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[24] J. Liu, M. Chang, and C.-K. Cheng. An Iterative Division Algorithm for FPGAs. In *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 83–89, 2006.

[25] Memory & Storage Architecture Lab@Seoul National University. SNU Real-Time Benchmarks. `http://archi.snu.ac.kr/realtime/benchmark`.

[26] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli. Valar: A Benchmark Suite to Study the Dynamic Behavior of Heterogeneous Systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 54–65, 2013.

[27] K. Morris. HLS versus OpenCL Xilinx and Altera Square Off on the Future. *Electronic Engineering Journal*, March 2013.

[28] G. Ndu, J. Navaridas, and M. Lujan. CHO: A Benchmark Suite for OpenCL-based FPGA Accelerators. Technical report, University of Manchester, May 2014.

[29] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193, May 2011.

[30] P. Panda and N. Dutt. 1995 high level synthesis design repository. In *System Synthesis, 1995., Proceedings of the Eighth International Symposium on*, pages 170–174, Sep 1995.

[31] D. B. Thomas, L. Howes, and W. Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 63–72, 2009.

[32] Xilinx.com. Sdaccel development environment. `http://www.xilinx.com/products/design-tools/sdx/sdaccel.html`, 2015.