A Unified OpenCL-flavor Programming Model with Scalable Hybrid Hardware Platform on FPGAs

Hongyuan Ding and Miaqing Huang
Department of Computer Science and Computer Engineering
University of Arkansas
Fayetteville, AR 72701, USA
Email: {hyding,mqhuang}@uark.edu

Abstract—Hardware accelerators are capable of achieving significant performance improvement. However, designing hardware accelerators lacks the flexibility and the productivity. Combining hardware accelerators with multiprocessor system-on-chip (MPSoC) is an alternative way to balance the flexibility, the productivity, and the performance. In this work, we present a unified hybrid OpenCL-flavor (HOpenCL) parallel programming model on MPSoC supporting both hardware and software kernels. By integrating the HOpenCL hardware IPs and software libraries, the same kernel function can execute as either hardware kernels on the dedicated hardware accelerators or software kernels on the general-purpose processors. Using the automatic design flow, the corresponding hybrid hardware platform is generated along with the executable.

We use the matrix multiplication of $512 \times 512$ to examine the potential of our hybrid system in terms of performance, scalability, and productivity. The results show that hardware kernels reach more than 10 times speedup compared with the software kernels. Our prototype platform also demonstrates a good performance scalability when the number of group computation units (GCUs) increases from 1 to 6 until it becomes a memory bound problem. Compared with the hard ARM core on the Zynq 7045 device, we find that the performance of one ARM core is equivalent to 2 or 3 GCUs with software kernel implementations. On the other hand, a single GCU with hardware kernel implementation is 5 times faster than the ARM core.

In this work we present a prototype of hybrid OpenCL-flavor (HOpenCL) parallel programming model, as well as a customized hybrid hardware platform on FPGA by providing a set of hardware and software libraries with the following features:

- A unified parallel programming model on FPGA with the capability of implementing parallelized software and hardware applications using state-of-the-art high-level synthesis tools [2].
- A more efficient memory hierarchy with features of DMAs and data caches.
- An automatic platform generation flow for both software and hardware HOpenCL kernels.

The current prototype of the HOpenCL platform consists of one global host processor and one compute device, which is comprised of several group computation units (GCUs). A GCU contains one local host processor and a couple of slave processors. The slave processors can be implemented as either general-purpose processors (e.g., MicroBlaze on Xilinx FPGAs) or hardware accelerators, which can be directly generated by high-level synthesis tools based on kernel functions.

The remainder of the paper is organized as follows. Section II introduces background and discusses related work on OpenCL and MPSoC on FPGAs. Section III introduces the hardware architecture of HOpenCL. In Section IV the software and hardware kernels are discussed, as well as the automatic system generation flow. In Section V we examine a benchmark to demonstrate the performance, the scalability and the productivity of the HOpenCL platform. We conclude this work and discuss some of our future work in Section VI.

I. INTRODUCTION

OpenCL [1] is a framework to design parallel applications on various computation resources (CPU, GPU, and FPGA). Programming using OpenCL consists of two steps. The first step is to define a computing platform on which the application will execute. In OpenCL’s term a platform consists of one host processor and multiple compute devices. The second step is to assign the computation tasks to each compute device and specify the dependencies among them through the explicit data transfer between these tasks. The information of both platform and the corresponding data affinity and parallelism is explicitly presented and easily extracted in OpenCL framework.

In recent years, FPGA devices have grown to a point where advanced processing technology is adopted to achieve higher frequency and capacity. Today’s FPGA is able to accommodate dedicated hard cores (including general-purpose processors, DSPs, and memories), as well as programmable logic on a single chip. Compared with general-purpose processors, hardware accelerators on FPGAs have the potential to gain higher performance since hardware circuits are more likely to take advantages of parallelism and other features in data flows.

However, due to the complexity of designing application-specific hardware accelerators, applications on FPGA usually lack the flexibility with a longer time to market. Implementing multiprocessor system-on-chip (MPSoC) with either vendor-provided or customized soft general-purpose processors and dedicated accelerators is an alternative approach to balancing flexibility and performance. The performance of a hybrid multiprocessor system mainly comes from the accelerators with the flexible programming on general-purpose processors.

II. BACKGROUND AND RELATED WORK

Due to the appealing feature of OpenCL in terms of architecture representation, it has been adopted in many related work to define the multicore architecture. In [3] a direct
implementation of OpenCL framework on Xilinx FPGA is presented. The OpenRCL machine consists of an array of processing elements, their on-board local memory, and an off-chip global memory. This work is generalized in the following work “MARC” [4]. In MARC (Many-core Approach to Reconfigurable Computing) an application is mapped to the MARC template, which consists of one control processor and 48 algorithmic processing cores. These 48 processing cores can be parameterized to fit the application requirements. The MARC approach is similar to the approach presented in [5], in which the authors developed a tool kit for embedded designers, including compiler, mapper, designers. The FlexCL approach proposed in [6] is used to configure the parameters of the open-source MicroBlaze-Lite software processor based on the application description. Chin and Chow introduced a memory infrastructure for FPGAs designed for OpenCL style computation [7]. An Aggregating Memory Controller is implemented in hardware and aims to maximize bandwidth to external, large, high-latency, high-bandwidth memories by finding the minimal number of external memory burst requests from a vector of requests.

RAMPSoC is a framework for generating an MPSoC system composed of multiple microprocessors and hardware accelerators for executing an algorithm [8]. An alternative to RAMPSoC is introduced in [9], which allows the runtime reconfiguration of heterogeneous processor cores with a finer granularity. In [10] OpenCL is used to design application-specific processors. Given an application written in OpenCL, an application-specific processor is generated to execute the application. In [11], the SOpenCL architectural synthesis tool is presented. The SOpenCL tool takes an OpenCL application and maps it to a custom designed hardware circuit. In this sense, it is still one variant of C-to-gate compiler, which is not the goal of this work. This approach is generalized in [12]. A similar approach is proposed in [13] in which OpenCL kernels are translated into CatapultC code for high-level synthesis. Altera has introduced its own OpenCL SDK [14]. Starting with OpenCL as an input, the SDK generates the host executable and the hardware accelerators that carry out the computation.

III. HARDWARE PLATFORM ARCHITECTURE

The current OpenCL specification is heavily influenced by GPU programming. In OpenCL a platform consists of one host processor and several compute devices, each of which contains multiple compute units. A single compute unit is comprised of multiple processing elements. An OpenCL function, called “kernel”, is assigned to one compute device during the runtime. A kernel function is implemented as a grid of work-items, each of which can be considered as a thread. The work-items in a grid are broken into work-groups, each of which is scheduled to execute on a compute unit. Every work-item is physically executed on a processing element. Through this mapping of compute device ↔ kernel, compute unit ↔ work-group, processing element ↔ work-item, the data parallelism in an application is explicitly expressed. On the HOpenCL platform, a kernel can run on either general-purpose processors or hardware accelerators. Then the kernel is called software kernel or hardware kernel, respectively.

Figure 1 demonstrates the memory model in HOpenCL in which all Group Computation Units (GCUs) combinely form a compute device. Similar to OpenCL, each processor in the GCU has its own private memory. All processors in one GCU share the same local memory. Global memory is visible to all GCUs. Global host processor is used to coordinate kernel execution and argument passing. Different from OpenCL, a specified local host processor in each GCU is assigned to coordinate data movement and work-group execution. Each GCU is also assigned with a local direct memory access (DMA), which is in charge of transferring block data from the local memory to the global memory and vice versa to avoid frequent bus requests to the global memory. DMA can be called only by the local host at the beginning of the execution of each work-group to avoid redundant data movement. Instead of having two separate global memories for the host and the compute device respectively in OpenCL, there is a union global memory shared by both the global host and all GCUs in HOpenCL. The advantage of a single global memory is to eliminate memory copies between the two global memories in OpenCL. Data requests to global memory are further optimized using caching. On the Xilinx Zynq 7045 device used in this work, a 512 KBytes cache as well as its cache controller are connected to the global memory utilizing the accelerator coherency port (ACP). All on-chip memory (OCM), cache, cache controller, global host and all GCUs are implemented on one Zynq 7045 device.

Figure 2 shows the structure of the basic HOpenCL hardware platform and the components inside each GCU, respectively, on the Zynq device. There are two levels of AXI buses, local bus and global bus, connecting local devices and global devices, respectively. Besides AXI buses, AXI-Stream is used to conduct operations that are not memory-mapped. Since AXI-Stream is a simple point-to-point connection, it does not waste AXI interconnection resources and increases no arbitration time. Inside each GCU, the local host is a general-purpose processor. Other slaves can be configured as either hardware accelerators for hardware kernels or general-purpose processors running software kernels. FIFOs are used to buffer asynchronous data flow. Other important components are discussed as follows.

![Memory model in HOpenCL](image-url)
Group Scheduler: Group scheduler provides a pair of numbers to map tasks on each GCU. Similar to OpenCL, where a problem space is divided into many N-dimensional workgroups, HOpenCL can support up to 2-D group size. Before proceeding to a new group, the local host will request a new group ID from the group scheduler, and assign it to each slave in its GCU. Since the number of GCUs is limited, the number of groups may exceed the number of GCUs when a large problem space is divided into many groups. Group scheduler will assign every group to each GCU following the principle of first-come, first-served (FCFS).

Group Number Allocator: In application developers’ view, all GCUs are symmetrical. Every GCU runs the same kernel program. However, in terms of hardware abstraction, every GCU needs to identify itself. In this case, a unique group number will be assigned by the group number allocator to each GCU. This number is received by the local host to achieve two objectives. The first one is to identify local-owned devices including DMA, local memory, and others that cannot be shared by other GCUs. The second one is to switch local daemon programs when there are hardware kernels running in the current group.

Global Status Memory: Global status memory (GSMem) stores a few synchronized signals and global shared information. Specifically, when trying to notify GCUs to start running kernels, the global host will write a trigger value to the associated locations in GSMem. Daemon programs running on local hosts will be polling these signals before executing kernels. In addition, the global host will write kernel arguments, which need to be passed to each GCU, into the GSMem.

Local Scheduler: Within every GCU, each slave processor and the local host can request a 2-D local ID from the local scheduler after a task is assigned to a GCU. HOpenCL supports up to two-dimensional problem size. Every ID request will be queued into the local scheduler with the principle of FCFS. In other words, every slave in one GCU has the equal opportunity to get a local ID. Those slaves running faster will get more IDs than those slower slaves.

Core Number Allocator: Since the local host does more jobs than other slaves, the local host needs to identify itself from others. Furthermore, once there are hardware kernels running in the current group, hardware accelerators and general-purpose processors will be identified with the core number allocated by the core number allocator.

Local Status Memory: Local status memory (LSMem) stores group running information. Since each DMA operation can only be carried out once for each work-group of work-items, after each DMA operation the local host will register this operation in LSMem to avoid executing it again. When noticing that the group ID generator FIFO is empty, the local host will write a value in LSMem to notify other slaves that all work-items in the current work-group have been finished.

Barrier: Barrier provides a hardware synchronization mechanism within each GCU. When a barrier request is received by a barrier function from any work-item in one group, other work-items cannot get the release signals from this barrier until they all send the barrier requests to the barrier. Once the release signals are obtained by work-items, they will continue proceeding.

IV. Hybrid Design Flow

Hardware accelerators can be implemented on FPGAs with EDA tools. However, designing hardware accelerators
TABLE I. OPENCL AND HOPENCL APIs.

<table>
<thead>
<tr>
<th>Execution Scope</th>
<th>OpenCL</th>
<th>HOpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>localId()</td>
<td>getLocalId()</td>
<td>getLocalId()</td>
</tr>
<tr>
<td>groupId()</td>
<td>getGroupID()</td>
<td>getGroupID()</td>
</tr>
<tr>
<td>globalId()</td>
<td>getGlobalID()</td>
<td>getGlobalID()</td>
</tr>
<tr>
<td>localSize()</td>
<td>getLocalSize()</td>
<td>getLocalSize()</td>
</tr>
<tr>
<td></td>
<td>barriert()</td>
<td>barriert()</td>
</tr>
<tr>
<td></td>
<td>simpleDMA()</td>
<td>simpleDMA()</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>setLocalSize()</td>
<td>setLocalSize()</td>
</tr>
<tr>
<td></td>
<td>setGroupSize()</td>
<td>setGroupSize()</td>
</tr>
<tr>
<td></td>
<td>cSetKernelArg()</td>
<td>cSetKernelArg()</td>
</tr>
<tr>
<td></td>
<td>KernelStart()</td>
<td>KernelStart()</td>
</tr>
<tr>
<td></td>
<td>KernelFinish()</td>
<td>KernelFinish()</td>
</tr>
</tbody>
</table>

that can achieve significant speedup compared with general-purpose processors is still a very tough job. This limitation impedes software developers using hardware to accelerate their programs. In order to ease this difficulty, FPGA vendors launched commercial high-level synthesis (HLS) tools that can generate hardware circuits from high-level languages. In our work, we develop both software and hardware HOpenCL libraries that can help users integrate their kernels into the HOpenCL platform.

Figure 3 demonstrates the generation of hardware platform and the flow of hybrid software design. Hardware configuration, including the number of GCUs, the size of local memory, and the configuration of hardware kernels, will be given as the input of the TCL script generator. Hardware platform is independent with software programming. In other words, how the hardware platform is configured mainly depends on the available hardware resources on the target FPGAs. Once the hardware platform is generated, the local scheduler as well as the global scheduler will try their best to maximize the performance of the applications running on the platform. Combining hardware platform, HOpenCL hardware IPs, and hardware kernels, the platform bitstream will be generated.

On the software side, users need to write OpenCL-flavor kernels and host programs with the associated APIs shown in Table I. HOpenCL provides essential functions inherited from

```c
#define _HOST_
#define _SLAVE_
#define _DMA_

#include "../HCL/hcl.h"
#include "malloc.h"

int main () {
   initCore();
   int A0 = 512;
   int A1 = 256;
   int B0 = 256;
   int B1 = 512;
   float *arrayA = (float *)malloc(A0*A1*sizeof(float));
   float *arrayB = (float *)malloc(B0*B1*sizeof(float));
   float *arrayC = (float *)malloc(A0*B1*sizeof(float));
   setLocalSize(16,16);
   setGroupSize(32,32);
   setKernelArg(0, (unsigned int *)arrayA);
   setKernelArg(1, (unsigned int *)arrayB);
   setKernelArg(2, (unsigned int *)arrayC);
   setKernelArg(3, (unsigned int *)A1);
   setKernelArg(4, (unsigned int *)B1);
   kernelStart();
   kernelFinish();
   return 0;
}
```

Fig. 5. Sample kernel code with DMA mode.

OpenCL. Different from OpenCL that provides an explicit method to manage kernel queue and context, HOpenCL users have to arrange kernel execution manually. Figure 4 shows a sample of how the global host passes arguments to GCUs and manages kernel execution. At the beginning, the global host needs to request a block of memory in the global memory. Since in HOpenCL the compute device and the global host share the same memory space, shared data (i.e., arrayA and arrayB) will not be moved between memories. The addresses

```c
#include "../HCL/hcl.h"
#include "malloc.h"

int main () {
   initCore();
   float *arrayA = (float *)malloc(A0*A1*sizeof(float));
   float *arrayB = (float *)malloc(B0*B1*sizeof(float));
   float *arrayC = (float *)malloc(A0*B1*sizeof(float));
   setLocalSize(16,16);
   setGroupSize(32,32);
   setKernelArg(0, (unsigned int *)arrayA);
   setKernelArg(1, (unsigned int *)arrayB);
   setKernelArg(2, (unsigned int *)arrayC);
   setKernelArg(3, (unsigned int *)A1);
   setKernelArg(4, (unsigned int *)B1);
   kernelStart();
   kernelFinish();
   return 0;
}
```

Fig. 4. Sample host code.
void single_hardware_kernel()
{
volatile unsigned int *data,
volatile unsigned int *barrierIn,
volatile unsigned int *barrierOut,
volatile unsigned int *coreNum,
volatile unsigned int *localID0,
volatile unsigned int *localID1,
volatile unsigned int *groupNum
}

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE m_axi port=data
#pragma HLS INTERFACE axis port=barrierIn
#pragma HLS INTERFACE axis port=barrierOut
#pragma HLS INTERFACE axis port=coreNum
#pragma HLS INTERFACE axis port=localID0
#pragma HLS INTERFACE axis port=localID1
#pragma HLS INTERFACE axis port=groupNum

Fig. 6. Hardware kernel interface.

of shared data are passed to GCUs through setKernelArg() and getKernelArg(). At last, the global host will trigger all
GCUs to start and wait to return until all GCUs finish the
current kernel. Figure 5 demonstrates a kernel program
of matrix multiplication with the option to enable DMA. Without
DMA, requests of data read and write will be arbitrated
through two levels of buses: the local bus and the global bus.
When DMA is enabled, data that are consecutive in global
memory and are accessed by the processors in each GCU
can be moved to local memory to reduce bus requests. There
are two implicit barrier operations in isCurrKernelFinish() and
simpleDMA(). Before proceeding to get new group IDs,
the local host obtains the group IDs from the group scheduler,
and then assigns them to other slaves later, before every slave
in one group can continue running. In simpleDMA(),
the local host registers DMA number in LSMem to guarantee
the one-time DMA operation in the same work-group. This will
avoid redundant data movement when the slaves try to get new
local IDs. Before the local host finishes DMA operations, all
slaves cannot release barriers.

HOpenCL also supports hardware kernel design in Vivado
HLS with the associated HOpenCL hardware libraries. Hardware
kernels are executed on hardware accelerators. Kernel
programs similar to software kernels can be converted into hard-
ware design by HLS tool. The differences between software
kernels and hardware kernels lie in the hardware interfaces
and the HLS design principles. Figure 6 shows the compatible
hardware interfaces with HOpenCL platform. An AXI master
interface is used to request data from buses. Six AXI-stream
interfaces are included to communicate with the local host
and other functional HOpenCL hardware IPs (i.e., barrier,
local scheduler, and core number allocator). When applying
hardware kernels, at least one general-purpose processor will
serve as the local host in each GCU to coordinate group
execution. Since HLS intends to unitize parallelism features to
optimize kernel program, sequential statements need to be han-
dled carefully. Figure 7 shows how barriers are implemented in
hardware kernel programs. Since there are no buffers between
barrier IP and hardware kernels, reading data from AXI-stream
will be blocked until the barrier IP releases the current barrier.
The IF statement guarantees that HLS will not optimize the
following statements; otherwise, barrier operations will not
work correctly.

unsigned int bIn;
+barrierOut = BarrierID;
bIn = +barrierIn;
if (bIn == BarrierID) {
  //Execution body after barrier operation;
}

Fig. 7. Barrier implementation in hardware kernel.

V. EXPERIMENTS AND RESULTS

In this section, we use the matrix multiplication for exper-
imental analysis to demonstrate the potential of our HOpenCL
platform. Experiments are conducted by using Vivado 2014.2
with the corresponding Vivado HLS. Hardware platform con-
figuration is shown in Table II. Through the AXI ACP shown
in Figure 2(a), the global bus is connected to the global
memory that runs at 533 MHz. The hardware platform itself
is driven by a 100 MHz clock. The inputs of our benchmark
are two 512×512 matrices. Table III lists the configurations
of the host and the kernel programs. Figure 8 shows how
the problem space of the matrix multiplication is divided into
groups and scheduled to GCUs. The whole problem space (i.e.,
the 2-D output array) is divided into groups. The numbers of
groups along the two dimensions are called the group size. The
numbers of elements along the two dimensions of the group
are called the local size. In this way, each item in a group
stands for one element of the output matrix. Once a group
is assigned to one GCU, all slave processors work together
to compute all elements in the group. Five different local
sizes, i.e., 2×2, 4×4, 8×8, 16×16, and 32×32, are tested. We
also test the effect of the DMA by enabling or disabling it.
Inside each GCU, the computation can be handled by either
general-purpose processors or hardware accelerators. Further
the number of GCUs can vary from 1 to 6. With the different
number of GCUs, local sizes, DMA modes, and kernel types,
we conduct the tests on total 120 combinations.

Figure 9 shows the FPGA resource utilization under 12
configurations. A and B stands for using MicroBlazes and hardware accelerators as slave processors, respectively. Compared with MicroBlazes, hardware kernels consume slightly fewer registers, LUTs, and BRAMs than MicroBlazes. Since hardware kernels are fully customized accelerators, more DSPs are used in order to maximize the performance by parallelizing computation.

The scalability of our HOOpenCL platform is expressed in Figure 10. For every number of GCUs, we measure the speedup for all five different local sizes with DMA enabled. The slave processors can be either MicroBlazes or hardware accelerators. When general-purpose processors are used as slaves, all 3 slaves plus the local host carry out the computation. On the other hand, when the hardware accelerators are implemented as the slaves, only the 3 hardware accelerators carry out the computation because they are much faster than the local host for the matrix multiplication. When the number of GCUs is fewer than 4, the speedup of both software and hardware kernels grows linearly as the number of GCUs increases. When the number of GCUs reaches 4 and above, the performance gain deviates from the linear projection. This trend is more obvious for hardware kernels. This deviation is due to the change of dominant factors that decide the performance of the system. The total execution time of the matrix multiplication benchmark consists of the computation time, the data movement overheads (when DMA is enabled), and the delay of bus requests. When there are less than 4 GCUs, the dominant factor of the total performance is the computation time spent by the processors. Since we use two-level buses with multiple memory hierarchies, the number of memory requests to the global memory are decreased by two AXI interconnections. Besides, the frequency of programmable logic (i.e., 100 MHz) on Zynq is configured much lower than that of the DDR interface (i.e., 533 MHz). It is difficult for the system to fully utilize the memory bandwidth when there are a few GCUs. However, when the number of GCUs increases, the dominant factor becomes the delay of bus requests. Hardware kernels are more likely to reach the maximum memory bandwidth. This is because dedicated hardware accelerators are better optimized for the data flows in the given tasks, and thus have more intensive memory accesses than the general-purpose processors.

Table IV extracts parts of results from Figure 10 where the local size is configured as $16 \times 16$ to demonstrate the speedup by enabling DMA with hardware kernels. Before starting new group execution, a block of consecutive data with the size of $512 \times 16$ from the first matrix will be fetched into the local memory from the global memory. No matter DMA is enabled or not, speedup by using hardware kernels is around 11 times. Speedup by using DMA in hardware kernels is slightly higher than that in software kernels. Hardware kernels are more sensitive with the distance of targeted memory than general-purpose processors since they usually have more intensive memory requests.

In order to further compare the performance potential of
TABLE IV PERFORMANCE RESULTS WITH THE LOCAL SIZE OF 16 × 16 (UNIT: s).

<table>
<thead>
<tr>
<th># of</th>
<th>Software Kernel w/o DMA</th>
<th>Hardware Kernel w/o DMA</th>
<th>Hardware Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUCs</td>
<td>w/o DMA</td>
<td>w/o DMA</td>
<td>w/o DMA</td>
</tr>
<tr>
<td>1</td>
<td>44.32</td>
<td>36.12</td>
<td>1.23</td>
</tr>
<tr>
<td>2</td>
<td>23.20</td>
<td>18.54</td>
<td>1.25</td>
</tr>
<tr>
<td>3</td>
<td>14.91</td>
<td>12.33</td>
<td>1.21</td>
</tr>
<tr>
<td>4</td>
<td>11.87</td>
<td>9.30</td>
<td>1.28</td>
</tr>
<tr>
<td>5</td>
<td>9.12</td>
<td>7.28</td>
<td>1.25</td>
</tr>
<tr>
<td>6</td>
<td>8.02</td>
<td>6.64</td>
<td>1.21</td>
</tr>
</tbody>
</table>

At this moment, we only support a limited set of original OpenCL APIs on our HOOpenCL platform. We plan to support more OpenCL APIs in the future development. In addition, we intend to improve the system design of the platform to further improve the performance, the scalability, and the ease of integration of hardware accelerators.

ACKNOWLEDGMENT

This work is supported in part by NSF under grant CNS-1219062. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The authors are grateful to Xilinx, Inc. for the donation of the FPGA boards and the software.

REFERENCES