Exploiting Hardware Abstraction for Hybrid Parallel Computing Framework

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

Abstract—Dedicated hardware modules on Field-Programmable Gate Arrays (FPGAs) are applied to achieve significant performance improvement over the software implementations. However, even in a simple system-on-chip (SoC) design on FPGA, hardware accelerators are managed explicitly by writing hardware/software interfaces. Programmers have to go through massive details to control and communicate with hardware accelerators, not to mention complicated functions to achieve memory allocation and hardware re-use. Besides, designing hardware accelerators still lacks productivity and flexibility. Promising data-level parallel applications can be accelerated by using recent parallel programming models. In this work, we extend software threads to hardware threads. By implementing a hybrid parallel computing framework the parallel applications are supported through the scheduling and reusing of the threads. Programmers can create software threads in an embedded OS environment without managing hardware details. Hardware manager takes over software threads and manage hardware resources in terms of reallocation and isolation. Experimental results demonstrate that hardware threads can achieve up to 10 times speedup over software implementation. The relationships between the length of scheduling time slices, and the performance and the response time are carefully evaluated. Besides, as the number of threads increases, the system demonstrates a good scalability.

I. INTRODUCTION

Although the virtualization of processor and memory resources is well developed in modern operating systems (e.g., Linux), for a long time hardware modules (including hardware accelerators) are managed as I/O devices [1]. Hardware engineers are required to design explicit data flow and command registers in order to interface with device drivers of the OS. Hardware accelerators are usually able to achieve significant performance improvement over processor-based software implementations. However, the inefficient management of hardware accelerators results in drawbacks to the whole system in two aspects.

Firstly, it is difficult for both software and hardware engineers to benefit from dedicated hardware accelerators. On one hand, software designers have to explicitly manage reallocation and isolation for memory and computing resources. On the other hand, due to the lack of appropriate programming models in hardware design, hardware engineers need to develop accelerators in lower levels, which usually causes a longer time-to-market and the worse quality of hardware systems. Secondly, computing resources (e.g., hardware accelerators) are wasted due to the lack of higher abstraction compared to processor resources in OS’ perspective. Most OS schedulers are designed to maximize the utilization of processors. But hardware accelerators may have more chances to be idle since no dedicated schedulers for hardware resources are designed in OS.

In this work, we manage hardware resources to adapt to parallel computing applications. Each parallel application is written in a kernel format comparable to OpenCL- or CUDA-like programming models, and is wrapped as a thread in Linux. Inside each thread, the application is decomposed and deployed on the hardware platform. From OS’ perspective, hardware resources are managed by parallel programming model and device drivers. On hardware platform, computing resources are refined as hardware threads (HTs). Hardware schedulers take over software threads created from OS and allocate available HTs to carry out computation. Besides, combining with hardware libraries, hardware threads can be generated from kernel programs with the help of HLS tools so that software engineers are released from complex hardware details.

The remainder of the this paper is organized as follows. In section II, backgrounds and related works will be discussed. Section III will focus on details about software implementation and hardware architecture. The proposed scheduling mechanism will be addressed in Section IV. In Section V our work will be evaluated and corresponding results are analyzed. We will conclude this work and discuss future work in Section VI.

II. RELATED WORK

There have been various works making efforts on abstractions of hardware computing resources. In BORPH [2], the hardware abstraction is achieved by adopting OS threads. OS programs along with hardware thread files are wrapped in BORPH objective files (BOFs). ReconOS [3], [4] extends software threads to hardware threads. A set of APIs are provided for software designers to communicate with hardware threads. They further support part of virtual memory management by providing memory management unit (MMU) for hardware threads. [5] also presents memory virtualization for reconfigurable hardware. HTThreads [6] leverages PThread programming models. A hardware scheduler is designed to support mixed scheduling for both software and hardware threads. [7] focuses on multi-threading by providing OS interfaces to manage hardware resources. All of these works extend software threads and provide intermediate layers for management of hardware threads in a software way. In our
work, we further extend parallel programming model on hardware resource abstraction and reallocation.

Other works on hardware scheduling have been proposed. In [8], a hardware-based context switching method is discussed to store and recover data on FPGA. [9] proposes hardware checkpointing by adopting the software one. [10] demonstrates execution policies for hardware threads. A mixed-architecture process scheduler that orients from OS native scheduler is presented in [1]. Hardware design from parallel programming models is proposed in [11], [12]. Altera has launched their OpenCL SDK [13]. By giving an OpenCL kernel program, the dedicated hardware platform will be generated automatically. Our work combines features from both hardware scheduling and parallel programming models. The original work-groups are extended to support context switching. Along with the scheduler for hardware threads, applications that are decomposed into multiple work-groups are executed and scheduled on hardware computing resources.

III. SYSTEM ARCHITECTURE AND SOFTWARE MODELING

Figure 1 demonstrates the system architecture on both software and hardware sides. The proposed system is built on Xilinx Zynq device consisting of processing system (PS) and programmable logic (PL). On PS part, there is a hard-copy ARM dual-core processor, on which a Linux operating system is booted. The hardware system is built on PL part. AXI general-purpose (GP) bus is used to provide command communication between PL and PS. Parallel applications are wrapped as host threads running on Linux. Various device drivers serve as both SW/HW interfaces and part of hardware management routines (e.g., interrupt handlers). Table I lists major device drivers used in our system. Since a complete embedded Linux development environment is provided, dynamic linkable libraries are comprised of both custom libs used in this system and standard Linux libs. On PL part, besides DMA that is used to move data between PS and PL DDR memories, the major functional components consist of the HT manager and multiple hardware threads.

**Hardware thread (HT) manager:** One of the most important jobs of the HT manager is to store and manage hardware thread description tables (HTDTs). Similar to the structure of `task_struct` that records process-related information in Linux, an HTDT is the description structure storing all the information of a thread when it is converted from a host thread to the hardware one. The information includes (1) application types that are used to determine which PR bitstream files are used to program on hardware threads; (2) application configurations that are passed to hardware threads from host threads; (3) Process ID (PID) and thread priority; (4) scheduling states that are used for storing checkpoints (e.g., current GID), and current state of hardware threads.

An HTDT queue is a linked queue storing all HTDTs. The HT manager keeps this queue as a FIFO queue. New registered HTDTs and HTDTs that are switched out but are not finished are added to the tail of this queue. A tail pointer will then move to point to the tail. The first available HTDT of the queue will always be fetched once schedulers send requests to the HT manager. Everytime when an HT scheduler tries to access the HTDT queue, a queue locker guarantees that

### Table I. Lists of Major Linux Device Drivers Used in the System.

<table>
<thead>
<tr>
<th>Drivers</th>
<th>Functions</th>
</tr>
</thead>
</table>
| sche    | (1) Communicate with HT schedulers via the HT manager.  
|         | (2) Change states of host threads passively. |
| dma     | Manage DMA to move data between PS and PL DDR memories. |
| reg     | Software thread register co-operating with the HT manager. |
| xdevcfg | Program PR and static bitstream files by using PCAP module. |
only one scheduler has the control of the HTDT queue at the same time. Once the scheduler fetches the head HTDT, the HT manager will change its state and notify the related host thread.

Hardware Threads (HTs): HTs consist of several hardware subthreads (HSTs) that are physical computation resources, the HT interface, and other supported modules. The HT interface coordinates its HT with the HT manager. After one scheduler gets an HTDT and stores it into its HTDT memory, the ID dispatcher will begin to assign group IDs and local IDs to conduct actual execution of the current thread extracted from this HTDT. A Local DMA can move data between PL memory and local memory for fast data accessing by HSTs. A Barrier inside each HT can provide synchronization for all HSTs in one HT.

A. Programming Model

In a parallel programming model such as 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. Programming with OpenCL needs two kinds of programs: host programs and kernel programs. The OpenCL kernel is assigned to one compute device during the runtime. Kernel functions are implemented as a grid of work-items, each of which can be considered as a thread-level execution. 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↔workgroup, processing element↔work-item, the data parallelism in an application is expressed.

Figure 2 demonstrates the memory model in our platform compared to the OpenCL one, as well as the execution hierarchy. The PS ARM processor serves as the host processor. All HTs with supported modules form a compute device. Each HT and its constituent HSTs map to the work-groups and the work-items, respectively. Similar to OpenCL, we provide a three-level memory hierarchy for compute devices including the global memory, the local memory, and the private memory. The global memory is shared by all computation resources on the platform. The local memory is provided to all HSTs inside an HT. The private memory is exclusively accessible by a particular HST. Besides, Local DMAs can move data between local memories and the global memory for faster accesses within each HT. The Global DMA outside of HTs is controlled by the host processor to move data between PS memory and PL memory (namely global memory in OpenCL model).

As shown in Figure 3, HTDTs in the HTDT queue are one-to-one mapped to host threads running on the host processor. Their unique PIDs are used to identify themselves. For one particular thread, the problem space is divided into multiple two-dimensional groups. All groups are of the same size. Mixed groups from all registered host threads are assigned to HTs. Different from software thread scheduling, in our system, a finer-grained scheduling (namely, group-level scheduling) are designed. In other words, each HT can execute groups from either the same or different threads. Each group has their own group IDs (GIDs) within on thread. Since GIDs are generated in orders the execution progress can be marked by using the GID. Then each group is divided into multiple items, which are the minimal processable elements physically deployed on HSTs.

Pseudocode of both host and kernel programs are listed in Figure 4 and Figure 5, respectively. At the beginning in a host program memory spaces storing the input and output data are allocated on PS and PL memories, respectively. Since the PL memory is not accessible by Linux on PS, the global DMA is used to move data between these two memories. DMA drivers are capable of allocating a piece of continuous memory space within kernel spaces of Linux. The physical address of this space along with its length are given to the DMA module. The virtual address in kernel spaces is mapped into user space of each host thread so that DMA memory spaces are fully accessible by user threads. During the DMA
typedef unsigned int HostAddr = (int *)hostMalloc(Size);
unsigned int KernelAddr = (int *)kernelMalloc(Size);
//Allocate memory space on PS and PL DDR.
DMAEnqueue(HostAddr, KernelAddr, size, dir);
//Put DMA commands on queue.
regThreads(PID, AppType);
setArgs(LocalSize, GroupSize);
//Register thread on hardware manager.
DMASleep();
//Wait for DMA queue clear.
while (!isFinished()) {
    schedulerSleep();
    //Wait for interrupts from hardware scheduler.
    if (isFinished())
        break;
    doPR(AvailableSlots);  
    //Program hardware threads on PR regions.
    }
DMAEnqueue(HostAddr, KernelAddr, size, dir);
unregThreads(PID);
//Clear HTDT on hardware manager.
DMASleep();
//DMA results back.

Fig. 4. Pseudocode of host programs.

for (i=0; i<length; i++)
    C[i][iid0,iidl] += A[iid0,i] * B[i,iidl];

Fig. 5. Pseudocode of kernel programs of matrix multiplication.

I/O waiting (multiple threads may access one DMA at the same time), the host program registers its current parallel application to the HT manager. The next while statement will exit when the current registered application is finished. Before that, its HTDT may switch out and in on different HTs for multiple times. Correspondingly, inside while statement, the host program may sleep, wakeup, and program PR regions on HTs for multiple times. Once finished, the host program will notify the HT manager to delete its HTDT in the HTDT queue and exit to join its parent thread on Linux.

On kernel programs, arguments (including addresses and sizes of the input and output data) are fetched from the HTDT memory in its HT interface. Then group IDs and local IDs generated by its ID dispatcher are assigned to variables. Then a unique pair of global IDs is computed from group and local IDs. The for statement takes matrix multiplication as an example to demonstrate how a single item is computed. Once the local memory is enabled, HSTs can access the frequent data faster by using nearer local memory instead of farther global memory.

B. Design Flow

Figure 6 shows the SW/HW co-design flow. Users are required to provide both host and kernel programs. In HW Generation step, the kernel programs are firstly interpreted to HLS source code that is adapted from C language. HLS tools synthesize kernel programs and export as checkpoint IP format. Combining with the provided hardware libraries written in HLS code and the platform static checkpoint file, bitstream files of PR regions along with the hybrid hardware platform are finally generated. In software generation step, the hardware description file is used as the configuration file for the Linux kernel. Then host programs, supported device drivers, and Linux kernel codes are cross-compiled to generate the bootable files. All bitstream files and Linux bootable files are stored on SD card for booting the system.

IV. HARDWARE RESOURCE REALLOCATION

The reallocation of hardware computing resources is implemented by using the hardware thread scheduling. In OS' perspective, thread states usually consist of states of ready, running, waiting, and exiting. The Linux implementation is shown in Figure 7(b). In our hardware thread implementation, as shown in Figure 7(a) the waiting state is eliminated because the hardware thread does not have the interruptible I/O waiting or the inner nested scheduling mechanism. On one hand, the possible I/O waiting of HTs comes from data accesses to
memories and the HTDT queue locker when schedulers try to get HTDTs from the HT manager. The former I/O waiting is too short when compared with overheads of context switching on HTs. Because one PR bitstream file for an HST is size of at least several hundred of kilobytes while the HST accesses data one word (4 bytes) after another. The interval time has very small chance to be longer than that of doing PR. The latter I/O waiting can be longer than the former one. However, since the group-level scheduling is leveraged hardware threads cannot be switched out before requiring an HTDT successfully. On the other hand, lower level scheduling than the group-level one is not necessary in a parallel computing framework like this one. Considering larger overheads of context switching in hardware threads than software threads, nested scheduling is not practical.

Figure 7(b) also describes the state transition of host programs running on Linux. Transition between the top two TASK_RUNNING states is managed by the Linux native scheduler, which does not affect our hardware scheduler routines. States of the host thread will change from TASK_RUNNING to TASK_INTERRUPTIBLE when various sleep() functions are called. In the device driver layer shared hardware resources (DMA, PCAP module, and HT manager) are designed to support multi-thread accessing. In other words, when a thread tries to access these resources that are already occupied by other threads, it will fall into TASK_INTERRUPTIBLE. When the host thread receives wakeup interrupts from hardware schedulers, its state will be changed back to TASK_RUNNING. As state transition of hardware threads shown in Figure 7(a), most state transition is between READY and RUNNING. When the scheduler acquires an HTDT, the corresponding HT will begin to run. After executing for a certain period of time, the HT will be switched out and go back to READY. Different from the software thread scheduler in Linux where execution is interrupted by timers (intervals between are called time slices), in our scheduler design, each HT is assigned with certain credits that are numbers of groups it can execute during one execution period. HTs are switched out when running out of credits.

The scheduling algorithms are demonstrated in Figure 8. HT schedulers need to firstly acquire the access to the HTDT queue before performing any operations (including fetching and updating) on HTDTs. Only during the time of one hardware thread conducting execution on its application, other scheduler may have chances to gain the control of the HTDT queue. The hardware thread will acquire next application from the HTDT queue when the current thread is fully finished or runs out of credits. After fetching the HTDT, the PR procedure can be ignored to save time if the previous application is the same with the current one. Otherwise, the new application (represented as bitstream file) has to be programmed on all HSTs within this HT. Once the current routine is finished or the PR is needed, the scheduler will wake up the host thread from sleeping mode (state of TASK_INTERRUPTIBLE). If the current application is switched out instead of finished, the scheduler will not only update hardware thread states, but also record the last group ID within executed ones to its HTDT. In this way states of a running HT are stored and will be recovered when this HTDT is acquired by one HT again. Since the minimal units that can be scheduled are groups, the number of credits is given in terms of the number of groups to execute within one routine.

V. SYSTEM EVALUATION AND ANALYSIS

We evaluate three micro benchmarks that will appear in embedded applications. They are listed as follows:

Matrix Multiplication: Two two-dimensional square matrices are used as the input data, and one 2D matrix will be computed as the output result. Different input sizes are examined. In order to map the problem with the parallel programming model, the output matrix is firstly divided into multiple 2D small matrices that are of the same size. These small matrices are assigned to every hardware thread.

Convolution: We preform a one-dimensional convolution. The input data is an integer array with different lengths. The filter size is of 128 elements. The output array is divided into different small arrays with their own group IDs.

Pre-scan: Addition is used as the operation. Input linear arrays are examined with different sizes. The output array has the same length as the input array. Each element of the output array is the sum of previous elements of the input array.

A. Experimental Setup

In this section, our work is evaluated through micro benchmarks under different configurations in two aspects: the performance and the average response time (ART). Experiments are conducted by using Xilinx Vivado 2014.4 with corresponding HLS tools on Xilinx ZC706 device. We leverage the Petalinux with Linux kernel version of 3.17 as our host operating system. Platform configurations are listed in Table II. The PL part is considered as one compute device compared with that in OpenCL programming model. We duplicate 4 hardware threads, each of which consists of 4 hardware subthreads. There are totally 16 HSTs, which means equal number of PR...
TABLE II. PLATFORM CONFIGURATIONS FOR EXPERIMENTS.

<table>
<thead>
<tr>
<th></th>
<th># of compute devices</th>
<th># of HTs</th>
<th># of HSTs in each HT</th>
<th>Sizes of PL and PS memories</th>
</tr>
</thead>
<tbody>
<tr>
<td># of compute devices</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1 GB</td>
</tr>
<tr>
<td>PS frequency</td>
<td>667 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL frequency</td>
<td>50 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Private memory is implicit since HLS will allocate storage space based on the source codes of kernel programs.

TABLE III. RESOURCE UTILIZATION OF HSTs IN PR REGIONS.

<table>
<thead>
<tr>
<th></th>
<th>Slice LUTs</th>
<th>%</th>
<th>Slice Registers</th>
<th>%</th>
<th>DSPs</th>
<th>%</th>
<th>Memory</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>1393</td>
<td>81.88%</td>
<td>1506</td>
<td>62.75%</td>
<td>20</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Prescan</td>
<td>1205</td>
<td>70.88%</td>
<td>1062</td>
<td>44.42%</td>
<td>4</td>
<td>20%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Convolution</td>
<td>1083</td>
<td>63.71%</td>
<td>978</td>
<td>40.75%</td>
<td>6</td>
<td>30%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>PR Region</td>
<td>1700</td>
<td>-</td>
<td>2400</td>
<td>-</td>
<td>20</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 9. Performance and the ART evaluated by setting various values of factor $F$.

Fig. 10. Average performance and the ART evaluated by creating different numbers of host threads.

regions are needed for each HST. Each of the three micro benchmarks is generated into 16 PR bitstream files in order to be available to place on all HSTs. Table III shows the resource utilization of micro benchmarks in PR regions. ZC706 device is equipped with two independent DDR memories, each of which is of 1 GB. They have two different memory interfaces. PL and PS memories are used for the compute device and the host processor, respectively. Host operating system is booted on PS memory that is divided into user and kernel spaces. Instead of sharing only one PS memory for both host and compute device, another extra PL memory provides more memory resources for parallelism applications running on the compute device.

On the host processor, each kind of micro benchmarks is wrapped as a host thread. Various numbers of micro benchmarks with different input and output data and weights are generated randomly to comprise a task pool. Once the OS is booted, a shell script will create every thread from the task pool one by one.

For schedulers, we use

$$Available\, \text{credits} = B + Weight \times F$$

(1)

to determine how many credits are assigned to each hardware thread. In this equation, $B$ stands for the base number of credits, which we set as 10 in our experiments. $F$ means the adjusted factor that determines the number of percentages of $Weight$ adding on the base credits. Various values of $F$ along with numbers of threads in the task pool are used to evaluate the performance and the ART on our system.

B. Results and Analysis

The performance is evaluated in terms of execution time. The average response time (ART) is the average time that an application needs to wait before executing on hardware threads. Before an application is actually finished, it may execute on hardware threads for multiple times. We take every period of waiting time to calculate the ART.

Firstly, the adjusted factor $F$ appeared in Equation 1 is changed from 0.5 to 2 with certain steps. Results of the system performance and the ART are demonstrated in Figure 9. Totally 32 host threads are carried out in the task pool. We categorize the task pool with different application types. The average weights of each kind of application might be different from each other. As shown in both Figure 9(a) and Figure 9(b),
when the factor increases, the performance will be better and the ART is going to be shorter. Examined into each figure, applications with larger average weights will also gain better results. Larger factor $F$ means longer time slices for hardware threads in each scheduling. For the performance evaluation, the result benefits from the fewer number of context switching of hardware threads. When hardware performs a context switching, usually a PR procedure will start, which is major overheads for scheduling. However, Figure 9(b) shows that different values of factor $F$ result in slightly changes of the ART. This is due to the relative weights between time slices of scheduling and the total execution time of each thread. Shorter execution time of one thread will result in more changes on the ART when factor $F$ increases.

In the second experiment the factor $F$ is fixed at 1, and the number of host threads in the task poll ranges from 2 to 64. The average performance and the ART is shown in Figure 10. The average performance is computed by dividing the total execution time by the number of threads. The average weight in this experiment is calculated by averaging the weights of all threads in the current task pool. As shown in Figure 10(a) the average performance is better when the number of threads is less than 8. The improvement comes from the elimination of PR overheads. When the number of threads continues to increase, the average performance is not affected. This property guarantees the good scalability of our system. This scalability results from the constant overheads of hardware schedulers when the number of threads grows because the scheduler can complete scheduling in $O(1)$ time. As shown in Figure 10(b) the ART goes almost linearly as the number of threads increases. There are negligible changes occurring on the ART when threads with different weights are running. This result is consistent with observations in Figure 9(b).

Besides, the speedup of hardware thread implementation over software ones is also evaluated. The Petalinux is running on both ARM cores. In order to fully utilize processors, we implement applications with two software threads running on two ARM cores together. When comparing with the application running as software threads, one single hardware thread is more than 2 times faster than software implementation. Four HTs can achieve up to 10 times speedup.

VI. Conclusion

In this paper, we try to abstract and manage hardware resources on FPGA to adapt parallel computing applications. One parallel application is conveyed as one hardware thread description that is managed by the HT manager. Applications are broken into multiple work-groups that are executed on hardware threads. A hardware thread consists of multiple hardware subthreads and hardware interfaces. Schedulers of hardware threads will allocate hardware resources for applications waiting at the HTDT queue. From software sides, parallel applications are wrapped into OS threads. In OS' perspective, once a thread is created the encapsulated hardware descriptions are registered on the HT manager. Our hybrid parallel computing framework then will take over the control of the execution of applications. We evaluate our work in terms of performance and the average response time (ART). When comparing with the software implementation, our hardware threads can achieve up to 10 times speedup. As the length of the scheduling time slice increases, the overall performance and the ART will become better. Besides, the system shows a good scalability when increasing the number of threads.

Current programming models usually leverage explicit memory allocation and isolation. In the future we will focus on memory abstraction on the parallel computing framework. Hardware and software threads can share data in a more implicit way on a lower OS kernel level.

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


