**Journal of Information Processing Vol.23 No.2 153–162 (Mar. 2015)**

[DOI: 10.2197/ipsjjip.23.153]

**Regular Paper**

# A Toolchain for Dynamic Function Off-load on CPU-FPGA Platforms

Takaaki Miyajima[1,a)] David Thomas[2] Hideharu Amano[1]

Received: June 30, 2014, Accepted: December 3, 2014

***Abstract:*** This new toolchain for accelerating application on CPU-FPGA platforms, called *Courier-FPGA*, extracts runtime information from a running target binary, and re-constructs the function call graph including input-output data. Then, it synthesizes hardware modules on the FPGA and makes software functions on CPU by using *Pipeline Generator*. The Pipeline Generator also builds a pipeline control program by using Intel Threading Building Block (Intel TBB) to run both hardware modules and software functions in parallel. Finally, Courier-FPGA's *Function Off-loader* dynamically replaces and off-loads the original functions in the binary by using the built pipeline. Courier-FPGA performs the off-loading without user intervention, source code tweaks or re-compilations of the binary. In our case studies, Courier-FPGA was used to accelerate a histogram-of-gradients (HOG) feature detection program on the Zynq platform. A series of functions were off-loaded, and the program was sped up 3.98 times by using the built pipeline.

***Keywords:*** design methodology, algorithm implementation, heterogeneous platform, FPGA

## 1. Introduction

Mixed CPU-FPGA platforms are often used in embedded processing for energy-efficient computing. They work by off-loading computationally intensive parts to a hardware module which is implemented in reconfigurable logic. To meet the performance requirements of recent advanced applications, legacy code working in embedded CPUs must be accelerated by reconfigurable hardwired logic. A large percentage of such legacy code uses popular function libraries like OpenCV. For such a function, either an optimized HDL design exists [1], or it becomes easier to generate a hardware module corresponding to each function by using recent high level synthesis tools for FPGA [2], [3]. On the other hand, the user sometimes cannot access the source code itself, or a pipelining method among multiple software hardware functions is not generalized.

We developed a tool chain, called *Courier-FPGA*, that analyzes the target binary running on the CPU, extracts information of functions and builds a function-level pipeline structure between the hardware modules on an FPGA and software functions on a CPU automatically. Unlike other researches on software-hardware co-synthesis or commercialized HLS tools, Courier-FPGA treats running binaries and accelerates them by replacing software functions with the built pipeline including pre-defined hardware modules. Courier-FPGA is based on our previous work, Courier [4]: an application accelerator toolchain for non-expert users. In our previous work, we described how Courier analyses the processing flow from the running binary and how it replaces the analyzed functions running on the CPU with corresponding non-pipelined functions of a GPU.

The followings are contributions of Courier-FPGA, and the differences from an original Courier:

- Original Courier is designed for a system with a host CPU and a GPU. In contrast, Courier-FPGA treats a CPU and multiple hardware acceleration modules implemented on an FPGA.
- By making the best use of the combination of CPU and multiple hardware acceleration modules, a mixed software hardware pipeline is introduced on CPU-FPGA platforms. *Pipeline Generator* builds the pipeline in which processing flow is the same as the original one even if the original flow is not pipelined.

We also conducted three practical case studies in which Courier-FPGA was used to make a mixed software hardware pipeline on Xilinx's Zynq platform. As a result, a binary of histogram of gradients (HOG) was sped up 3.98 times on the existing hardware modules. Two other cases were also sped up 22.1 times and 1.29 times, respectively.

The rest of this paper is organized as follows. In Section 2, we overview Courier-FPGA, including its features designed for making function call graphs including input-output data from the running binary and off-loading. Section 3 describes the details of mixed software hardware pipelines on CPU-FPGA platforms. Section 4 gives case studies showing the capability of Courier-FPGA. We discuss our proposal and related work in Section 5. Finally, we conclude the paper.

## 2. *Courier-FPGA*

*Courier-FPGA* is based on Courier, a toolchain for a single accelerator like GPU. In this section, we start by giving an overview

[1] The authors are with the Graduate School of Science and Technology, Keio University, Yokohama, Kanagawa 223–8522, Japan
[2] The author is with the Department of Electrical and Electronic Engineering, Imperial College London, United Kingdom
[a)] vision@am.ics.keio.ac.jp

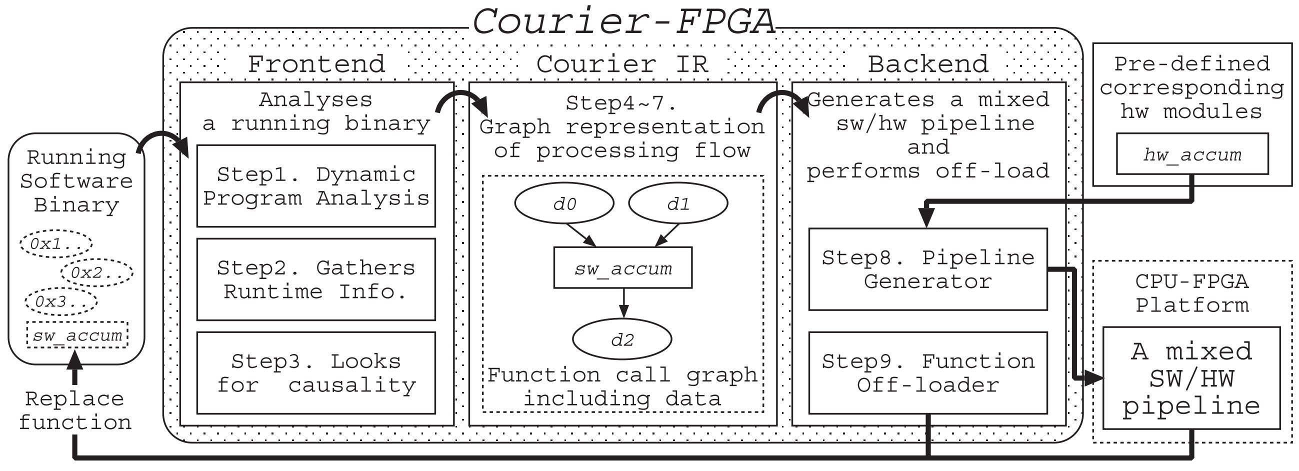

Courier-FPGA
Frontend
Courier IR
Backend
Pre-defined corresponding hw modules
hw_accum
Running Software Binary
0x1..
0x2..
0x3..
sw_accum
Replace function
Analyses a running binary
Step1. Dynamic Program Analysis
Step2. Gathers Runtime Info.
Step3. Looks for causality
Step4~7. Graph representation of processing flow
d0
d1
sw_accum
d2
Function call graph including data
Generates a mixed sw/hw pipeline and performs off-load
Step8. Pipeline Generator
Step9. Function Off-loader
CPU-FPGA Platform
A mixed SW/HW pipeline

**Fig. 1** Overview and work-flow of Courier-FPGA: the *Frontend* analyzes the running binary (Step1, 2 and 3) and then generates a function call graph including input-output data and a *Courier Intermediate Representation* (*IR*). The user refers to the graph and results, decides which parts to off-load and rewrites IR if needed (Steps 4, 5, 6 and 7). After that, the *Pipeline Generator* builds a mixed sw/hw pipeline (Step 8). Finally, the *Function Off-loader* replaces function and off-loads it to the accelerator (Step 9).

of Courier. Then we describe the detail of Courier-FPGA and its features to make the best use of CPU-FPGA platforms.

## 2.1 Overview of Courier

Recent application programs use some open-source libraries such as OpenCV or BLAS. Users who want to accelerate them may not know enough about the source code. Sometimes, they cannot access the source code and only have the executable binary. However, the conventional work-flow of accelerating application programs is designed for expert programmers who have developed the program on the CPU. As a result, it can not be utilized by non-experts to accelerate such programs. On the other hand, enough optimized corresponding functions of such open-source libraries are available for popular accelerators like GPUs and FPGAs.

A motivation of Courier is to provide a simplified work-flow of application acceleration for non-expert users. Courier requires a target binary and pre-defined corresponding functions of the accelerator. A user only designates a running target binary to Courier. Courier starts analysis and then constructs a processing flow of the binary. The functions in the binary which is running on the CPU can be dynamically and automatically replaced with the corresponding functions of the GPU. "Original" Courier is capable of application analysis, processing flow graph construction and dynamic function replacement. It cannot build function-level pipelines nor deal with hardware modules on an FPGA. By providing *Pipeline Generator* and *Dynamic Off-loader*, *Courier-FPGA* can build a mixed software hardware (sw/hw) pipeline on a CPU-FPGA platform.

**Figure 1** illustrates an overview of *Courier-FPGA* and its work-flow. Frontend and Courier IR of Courier-FPGA are the same as those of original Courier, but Backend newly supports FPGA. Courier-FPGA is comprised of three main parts: *Frontend, Courier Intermediate Representation* (*IR*)*, and Backend.*

- The *Frontend* analyzes a running target binary and takes a heuristic approach to make the function call graph including input-output data from gathered information. The Frontend doesn't require access to the original source code or any sort of re-compilation. It can recognize the functions in the graph to be the targets of acceleration. Moreover, it can refer to the input/output data and properties of them in the graph during the acceleration process to decrease the number of communications between the CPU and accelerator.
- *Courier Intermediate Representation* (*IR*) is a simplified language that enables users to modify dataflow and designate functions to off-load to the Backend if needed.
- The *Backend* automatically off-loads the function, if the corresponding function is ready for the accelerator. The *Function Off-loader* automatically decreases the number of communications along with off-load, and maintains the original processing flow before and after off-load. Original Courier can deal with GPU, while *Courier-FPGA* can deal with FPGA.

The situation of the figure is as follows: there are a target binary which is running on a CPU and "hw_accum" that is a pre-defined corresponding function for accelerator. "hw_accum" is required by Courier-FPGA in order to achieve shorter processing time. If there's no corresponding module, Courier-FPGA builds a pipeline which has only software functions. Frontend finds that the binary executes a processing flow, "sw_accum" function which obtains two input data (0x1.. and 0x2..) and produces one output data (0x3..). Then "sw_accum" is replaced with the "hw_accum" and off-loaded. The caption of the figure describes the work-flow of Courier-FPGA. The user can refer to the function call graph including input-output data and modify it in Courier IR interactively.

## 2.2 Frontend

The *Frontend* is composed of three main steps so as to make the *Function call graph including input-output data*. This graph includes the chronological order of function calls, their input/output, and profile data. We used dynamic program analysis and a heuristic approach. Users simply start their application as usual, the Frontend can analyze functions/data whose function definition/data type information are known in advance, and makes the graph during execution. Each step works as follows.

Step 1. The Courier traces the running binary by using a tracing program,
Step 2. gathers runtime information during execution,
Step 3. and looks for the causal function call including input-output data.

Note that the Frontend of Courier-FPGA is part of the original Courier, and details are described in our previous paper [4].

### 2.3 Courier Intermediate Representation (IR)

*Courier IR* is an intermediate representation that enables users to modify the processing flow and designate parts to off-load to the Backend. Thanks to fully automated Frontend and Backend, users can refer and choose off-load parts by using this IR if needed. The two main processes of Courier IR are as follows. Note that at present, Courier IR is manually translated from the graph.

Step 4. Courier generates an IR corresponding to the processing inside binary and
Step 5. generates the function call graph including input-output data, and
Step 6. the user examines the graph, and then
Step 7. modifies the processing flow or designates off-load parts if needed.

The Courier IR of Courier-FPGA is part of the original Courier, and details are described in our previous paper [4].

### 2.4 Backend

The *Backend* is designed for automatic off-loading at runtime, and consists of two steps as follows.

Step 8. The *Pipeline Generator* builds a mixed sw/hw pipeline. It first generates the corresponding hardware module on an FPGA, and then prepares software functions and a pipeline control program.
Step 9. Finally, the *Function Off-loader* selects a path and replaces functions with the generated pipeline.

The main functions of the Backend are the *Pipeline Generator* and *Function Off-loader*, which were originally developed for Courier-FPGA. Their details are explained in the next section.

## 3. Dynamic Function Off-load System for CPU-FPGA Platforms

### 3.1 Fundamental Concept

After the Frontend analyzes the running binary and makes a function call graph including input-output data, the Backend automatically builds a mixed sw/hw pipeline and off-loads the functions to the pipeline. The pipeline includes pre-defined corresponding hardware modules on an FPGA if they exist. If a function does not have a corresponding hardware module, it is run only on CPU. Hence, the extracted flow is divided into tasks and

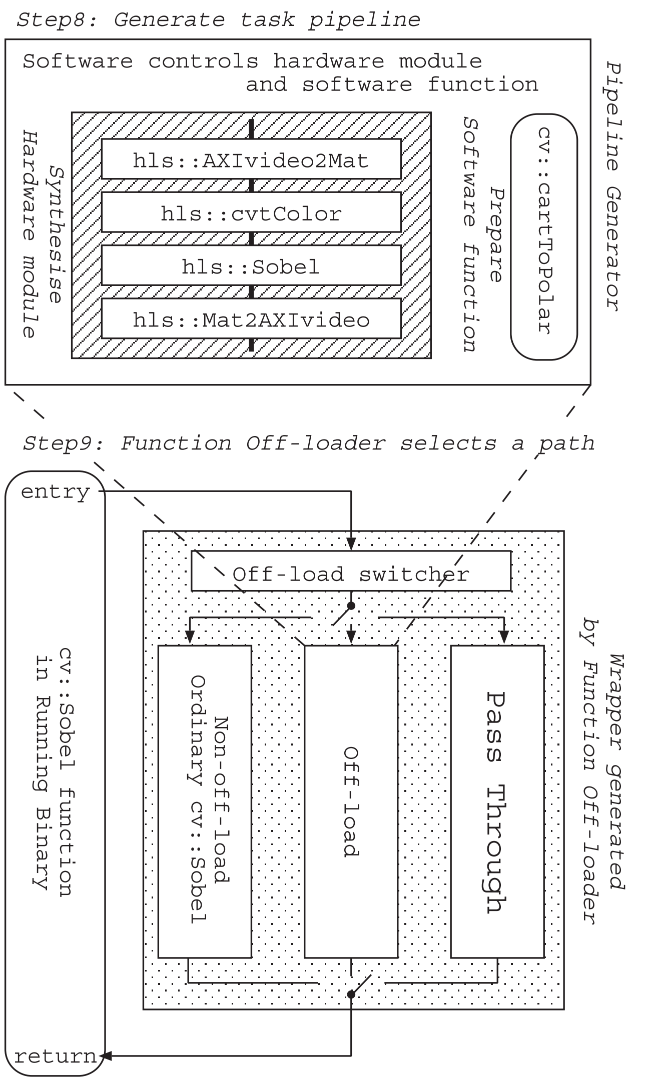

Step8: Generate task pipeline
Software controls hardware module and software function
Pipeline Generator
Synthesise Hardware module
hls::AXIvideo2Mat
hls::cvtColor
hls::Sobel
hls::Mat2AXIvideo
Prepare Software function
cv::cartToPolar
Step9: Function Off-loader selects a path
entry
Off-load switcher
cv::Sobel function in Running Binary
Non-off-load Ordinary cv::Sobel
Off-load
Pass Through
Wrapper generated by Function Off-loader
return

**Fig. 2** In step 8, the Pipeline Generator prepares a mixed software hardware task pipeline. In step 9, the Function Off-loader uses the pipeline when "off-load" is selected.

each task is composed of multiple software functions or hardware modules. Here, a "task" is not a "fine grained" calculation such as a single x86 assembly code or arithmetic operation on an FPGA, but a process with a certain amount of computation, such as a group of a few functions [5]. Unlike a single GPU, the off-loading target of original Courier, the target is multiple tasks than can work in parallel. Both software and hardware tasks should run in a pipelined manner so as to make the best use of the parallelism.

**Figure 2** shows an example of dynamic off-loading by using the *Pipeline Generator* and *Function Off-loader*. The Structure of a built pipeline; a mixed sw/hw pipeline on a CPU-FPGA platform, is composed of the following three main parts:

- A task pipeline control program: Program that runs the software and hardware tasks in parallel.
- Software task: Software functions run on the CPU.
- Hardware task: Hardware modules run on the FPGA.

The top panel illustrates Step 8, in which the *Pipeline Generator* makes a mixed sw/hw pipeline. First, the Pipeline Generator automatically generates a code of pre-defined corresponding hardware modules, configures them on the FPGA, and prepares

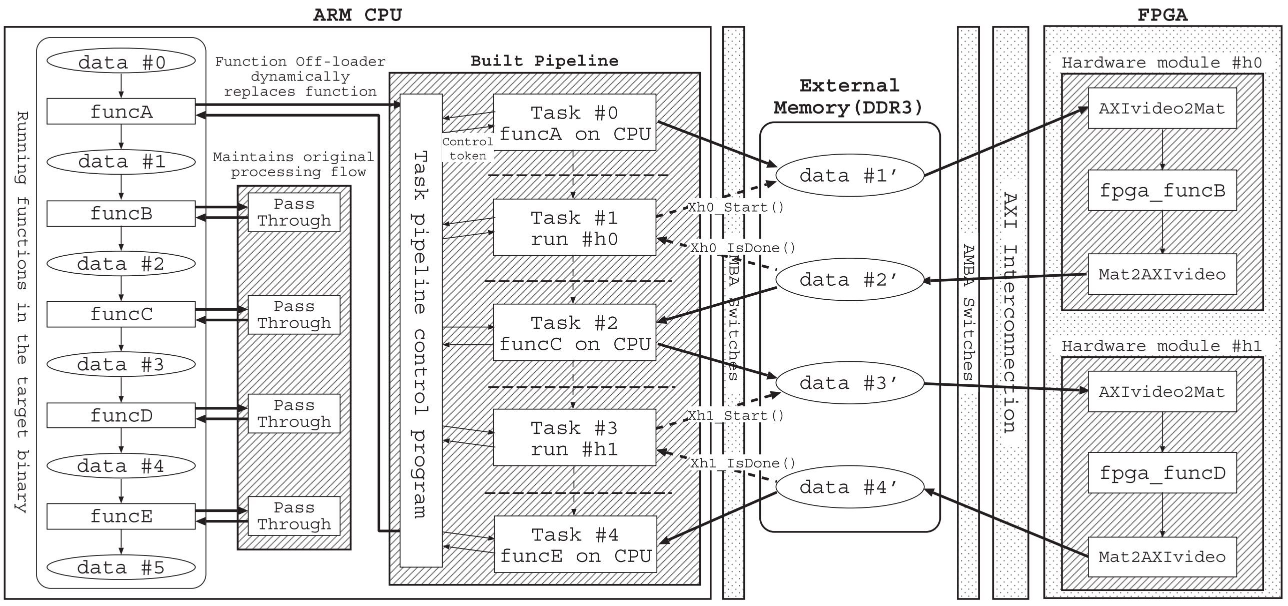


**Fig. 3** Behavior of a typical mixed software hardware pipeline controlled by software program. Shaded rectangles are generated by Pipeline Generator. Tasks run in a pipelined manner, and each task can send and receive input and output data which is indicated by bold line. Input and output data of tasks are stored in the external memory. In this case, Task #1 and #3 run hardware module #h0 and #h1 while Task #0, #2 and #4 run software function.

software functions. Then, it makes a control program that runs mixed software hardware tasks in parallel. The parallel tasks perform processing corresponding to a target binary.

In Step 9 that is illustrated on the bottom of Fig. 2, "cv::Sobel" in the target binary is replaced with a wrapped function which is made by the *Function Off-loader*. The wrapped function includes a switcher. When "off-load" is selected, the control program made by the Pipeline Generator in Step 8 starts the process. Even if the functions in the target binary run sequentially, the *Function Off-loader* can perform the same processing in a pipelined manner by using the built pipeline. **Figure 3** shows a typical case of building a mixed sw/hw pipeline.

In this section, we describe how the Pipeline Generator automatically builds an efficient mixed sw/hw task pipeline, and how the Function Off-loader performs off-loading dynamically.

## 3.2 Software Controlled Task Pipeline

A task pipeline control program that runs mixed software hardware tasks in a pipelined manner is needed in order to maximize the processing power of a CPU-FPGA platform. Recently, platforms such as Zynq [6] and Arria V SoC [7] have emerged which integrate FPGA and ARM CPU. In addition, there are some open source libraries to enable parallel execution on the ARM CPU, for example pthreads [8], Boost::thread [9], OpenMP [10], or Glib::thread [11]. However, they are not intended for pipelined execution of tasks.

Intel Thread Building Blocks (TBB) is a flexible open source library that runs multiple functions in a pipelined manner on a multi-core CPU. The *tbb::pipeline* class is provided to build a straight forward pipeline. A user adds an arbitrary task to each stage of the pipeline skeleton, and also specifies the processing order and a parallelism of the stages. After that, TBB automatically runs the tasks in a pipelined manner. TBB introduces the concepts of a thread pool and token base pipeline. Multiple slave threads are managed by a master thread. Master thread assigns a task which is registered by the user to an idle slave thread and also transfers input data. Then, the slave thread runs the task and finally sends back output data and a token to the master thread. TBB is also capable of double buffering when two or more tasks are registered. This type of pipeline makes it easy to re-order and insert new tasks.

Figure 3 shows a behavior of a typical mixed sw/hw pipeline controlled by a task pipeline control program. The Pipeline Generator first searches for corresponding hardware functions to replace the running functions in the target binary, which is illustrated on the left of the figure. To find appropriate hardware modules, we create a table which contains correspondence relationship between software functions and hardware modules. Pipeline Generator searches corresponding modules from the table and uses registered modules. In the case of *cv::sobel* function in OpenCV library, a corresponding hardware module is *hls::Sobel*. A user can add correspondence relationship of user-original modules to use them. In the case of the figure, Courier finds two corresponding hardware functions: func B and E. Then, it generates source code of two hardware modules: the former contains fpga_funcB, and the latter contains fpga_funcD. In addition, Task #1 and Task #3 which just send and receive input and output data are also generated as a software part. On the other hand, there are no hardware modules for funcA, C and E, so software functions is made for them. Thus, five tasks, two hardware modules and three software functions, are generated for the five pipeline stages shown in the figure. Tasks are individually compiled as a shared object before a deployed run. The pipeline control program runs these tasks in parallel.

On a deployed run, tasks work as follows from the viewpoint of the target binary. The Function Off-loader hooks and replaces funcA with Task #0. It also hooks first input data (data #0) from the running binary. Then, Task #0 first executes dynamically loaded funcA and stores the result (data #1') in external memory. And then, Task #1 invokes "start command" (Xh0_Start()) to send the data to the hardware module #h0, and receives "done signal" (Xh0_Done()) when fpga_funcB finishes a process and stores a result data (data #2') in the memory. While Task#1 is processing the first data, the pipeline control program starts Task#0. Consequently, the second input data from the running binary are simultaneously processed by Task#0. This is a software controlled task pipeline. Note that intermediate data such as "data #1'" are stored in the external memory and data start/done commands are automatically generated by Xilinx's high-level synthesis tool.

Unlike a common hardware pipeline in which the previous stage cannot start until the next stage has finished, a pipeline provided by TBB can start each stage even if the next stage doesn't finish. For example, Task #0 can take the second input while Task #1 is processing a time consuming task for the first input. As a result, the pipeline can reduce the probability of stall compared with the hardware pipeline. Additionally, stages which run in parallel can be dynamically changed since a task is randomly assigned to an idle thread by the control program.

## 3.3 Building an Efficient Mixed Software Hardware Pipeline

When we build a mixed sw/hw pipeline, we have to consider the following items in order to make it efficient. It is equal to a decision of which tasks can run in parallel and how to divide the extracted flow into some stages. We proposed the following solutions and implemented them in the Pipeline Generator so as to automatically generate an efficient pipeline.

(1) Concurrency: the concurrency of each stage.
(2) # of threads: the number of threads which run in parallel.

### 3.3.1 Concurrency

A feature of the mixed sw/hw pipeline is that stages which run in parallel can be dynamically changed. In typical video processing, only the image input/output must run serially, while the rest of the function can run in parallel. The former is parameterized as *serial_in_order*, and the latter is parameterized as *parallel* in TBB. The Pipeline Generator defines the *volatileInput/Output* as *serial_in_order* so as to make them run in sequential and the rest of the functions as *parallel* so as to make them run in parallel by default.

### 3.3.2 Number of Threads

The number of tasks which can run in parallel, depends on the number of logical threads on the platform. The number must be defined to make the task pipeline with TBB. The Pipeline Generator automatically sets the parameter to the maximum number of threads in order to build an efficient pipeline control program. In the case of Xilinx's Zynq, there are two logical threads. It means that even if there are many tasks, only two tasks can run in parallel. This limitation will be relaxed in future embedded CPU cores which can run more logical threads. For example, the quad-core ARM Cortex-A7 is already available. When we use this quad-core CPU, four tasks can run in parallel.

Current Pipeline Generator divides the extracted processing flow into some stages by using the simple partitioning policy: "Pipeline Generator divides total processing time by the number of threads plus one and searches the closest sub-total of processing time of functions". It can be formulated as follows.

$$T_{stage} = T_{total} \div (N_{logical\_thread} + 1) \qquad (1)$$

Where $T_{stage}$ is a target time of each stage, $T_{total}$ is a total processing time, $N_{logical\_thread}$ is the number of logical threads and $N_{logical\_thread} + 1$ is the number of pipeline stages. The policy is derived from the following considerations. According to our preliminary evaluation, the number of stages should be close to that of a logical thread of the Zynq because controlling many tasks is a heavy job for Zynq's CPU. Furthermore, to keep the minimal processing time, each pipeline stage should run in nearly the same time, i.e. a balanced pipeline. Note that, processing time of software functions can be obtained in the analyzed data from the Frontend and that of hardware modules can be estimated by the logic synthesis tool, and thus processing time of all functions are available at this time.

## 3.4 Generating a Code of Hardware Module

For each hardware task in Section 4, we used an OpenCV-compatible high-level synthesis library provided by Xilinx [2]. The Pipeline Generator generates the source code of the hardware module of corresponding processing, and adds an input/output port for the module. The AXI4-Streaming protocol [12] and Video DMA controller are used for the input/output port to communicate with the ARM CPU and the hardware module. *AXIvideo2Mat* and *Mat2AXIvideo* are added in a source file so as to synthesize the ports and the DMA module. In the case of a mixed sw/hw pipeline, intermediate data are stored in external memory. Thus, input data from the software is first stored in the DDR3 on-board RAM on Zynq before being processed and stored again in the RAM after processing. This kind of streaming architecture requires to read and write the data into the DDR3. Hence, the bus width of the input and output port significantly influences the performance. To deal with this problem, current Pipeline Generator automatically calculates and defines the width of the port by using the extracted bit-depth information from the Frontend. Furthermore, the Pipeline Generator tries to pipeline a series of functions if the functions have no branch nor loop. This pipelining is performed by inserting *#pragma HLS STREAM* in the head of the generated functions. Finally, generated codes are synthesized and placed on an FPGA. In addition, Courier-FPGA can use user-defined hardware modules if they have AXI-Streaming ports and are integrated into Zynq. But it doesn't have any kind of automatic port generation mechanism or automatic integration mechanism currently. Programmers must manually add the AXI ports to the user-defined modules and integrate the modules into the platform when they want to append them for off-loading.

Generated hardware modules are prepared as a block device, and basic device driver APIs are prepared by Xilinx's high-level synthesis tool. In the case study, *XTask0_Start()* function sends input data to start the process on the hardware module, and *XTask0_IsDone()* function polls done signal until the hardware module finishes a process. These API functions are used in a task on the CPU side.

### 3.5 Off-loading Tasks

The *Function Off-loader* in the Backend automatically makes a function wrapper to replace the original function designated by Courier IR. The wrapper contains the equivalent accelerator function that is built by Pipeline Generator including a pre/post-processing and data transfer. This mechanism of Step 9 behaves as follows before the run is being deployed. Courier-FPGA stops the running binary when Step 8 finishes, and then the Function Off-loader intercepts (hooks) the designated functions. It then replaces the original functions with the wrapper that includes the *Off-loader Switcher* and a software task. The Function Off-loader maintains processing flow and optimizes the data transfer by choosing one of the three paths of the Off-load Switcher. Finally, Courier-FPGA re-starts the binary. This process does not require any user intervention.

An example wrapper is shown at the bottom of Fig. 2. The wrapper has an *Off-load switcher* that provides one of three possible paths for a function: *non-off-load*, *off-load*, and *pass through*. Each path selected by the Function Off-loader works as follows.

- *Non-off-load* keeps the same function as the original, so the function runs on the CPU.
- *Off-load* replaces the designated function with the software program generated by the Pipeline Generator, and the task pipeline starts the process.
- *Pass Through* assigns the input data directly to the output data so as to skip the function in binary.

In order to reduce the number of data transfers and stages in the pipeline, the Function Off-loader replaces "the head" of a series of functions. Multiple functions run and are pipelined by using the built pipeline. In Section 4, functions in the target binary are divided into four tasks and run in a pipelined manner. The Function Off-loader hooks *cv::cvtColor*, and the series of designated functions are also executed here. And the rest of the functions are replaced with *Pass Through*. By using the Function Off-loader, the number of data transfers is optimized and an efficient pipeline can be built. To maintain the original processing flow, successive functions must be passed in the original binary running on the CPU. Thus, the Function Off-loader replaces and skips them by using *Pass Through*.

## 4. Case Study

In this section, we illustrate our work-flow by describing three practical case studies. The experimental conditions were as follows: the running binary was analyzed on Fedora 20 (Kernel 3.14.3-200.fc20.x86_64), The binary was deployed on Zynq-7000 AP SoC (XC7Z020-CLG484-1) on Zedboard. Zynq-7000 was composed of a Dual Core ARM Coretex-A9 CPU 667 MHz with 512 MB memory (called PS: Processing System) and 85,000 Series-7 programmable logic cells (called PL: Programmable Logic). Linaro 32 bit (Debian 7.0) ran on the PS. We also used Xilinx Vivado HLS and Vivado 2014.2 as a synthesis tool.

### 4.1 Histogram of Oriented Gradients (HOG)

HOG is a widely used feature detection algorithm for purposes such as face recognition [13], and OpenCV is a widely used open software library for computer vision [14]. The HOG implementation in OpenCV includes main features that are commonly seen in computer vision applications: OpenCV C++ API functions (e.g., Sobel operator) and diverging/converging flow.

The processing flow of the HOG algorithm in the running binary consists of the following four main steps.

[Step #0] Apply the Sobel operator: Each frame of the video source is converted into gray scale by *cv::cvtColor*. Then, x- and y-axis Sobel operators (*cv::Sobel_x, cv::Sobel_y*) are applied. Both operators obtain the same gray scale image.

[Step #1] Compute the gradient and magnitude: The gradient and magnitude are calculated from the x/y Sobel images by using *cv::cartToPolar*.

[Step #2] Adjust gradient: gradient values are adjusted to within 0 to 180 degrees by *cv::threshold* and *cv::subtract*. The two images generated in Step #2 are combined into one image (*cv::add*), and adjusted gradient values are calculated.

[Step #3] Create histogram: Lastly, the image is divided into a nine-channel histogram by using *cv::divide*.

### 4.2 Acceleration Work-flow of Courier-FPGA

***I. Analyze running binary***

After the user designates the running target binary, the Frontend of Courier-FPGA analyzes the running binary, then it generates the graph and IR. This process step corresponds to Steps 1∼3 in Fig. 1. The Frontend extracts the following runtime information during the profile run:

- OpenCV C++ API function name with arguments,
- function start/end absolute time (execution time),
- # of input/output of functions,
- raw value of input/output image data, and
- image properties (size, bit depth, and channels).

***II. Generating the function call graph including input-output data of the running binary***

After the profile run, a function call graph including input-output data of the running binary is automatically generated (see the left of **Fig. 4**). The user examines the graph and decides whether to off-load and non-off-load parts if needed. The graph is identical to the previously described processing flow. Ellipse nodes and rectangle nodes represent images and functions, respectively. The size of the node reflects the execution time or the size of the data (height × width × bit-depth × channels; e.g., the first node is 1,280 × 720 × 32 bit × 1-channel). The processing time is shown in the second row of the ellipse node. Nodes are aligned in chronological order. According to the graph, each input image is 1,280 × 720 and processed in 650,856 [$\mu$s] in total. This is less than 1.5 frames per second ([fps]).

The Courier IR description is automatically generated. Users can modify this to change the actual processing flow if needed. The details of the IR in the case study have been omitted because of space limitations.

***III. Acceleration***

In this step (Steps 8 and 9 in Fig. 1), Courier-FPGA first searches for "safely off-loadable" parts, where a processing flow is straight-forward, functions and input/output data are both

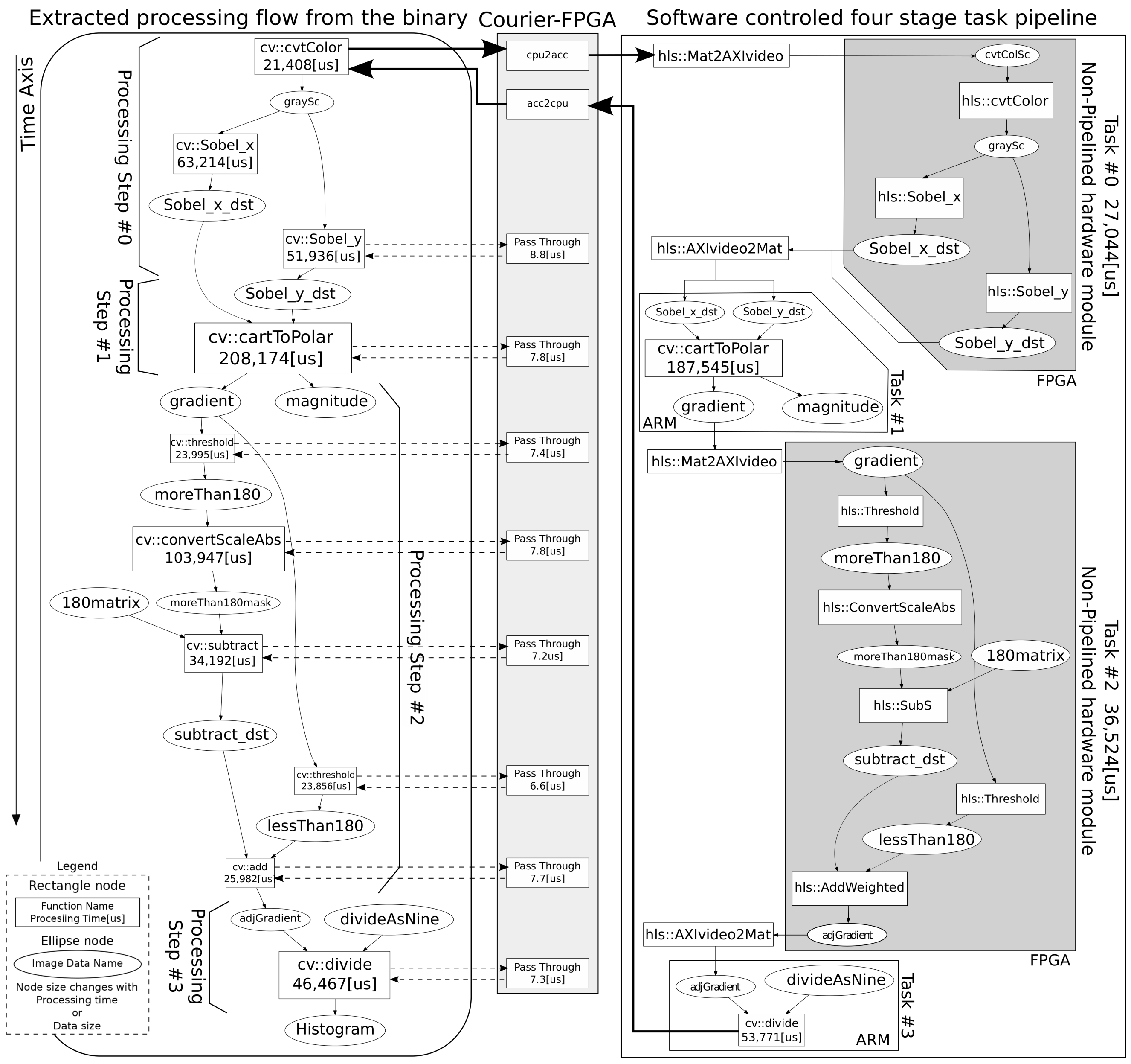


**Fig. 4** Processing flow extracted from the running binary (left) and off-loaded flow (right). Each processing step is assigned to a task. The Function Off-loader generates a four stage mixed software hardware pipeline.

traced, and a corresponding accelerated hardware module is available. For such parts, Courier-FPGA automatically builds a mixed sw/hw pipeline by using the Pipeline Generator and off-loads it by using the Function Off-loader in default mode.

In this case, the Pipeline Generator generates a four-stage mixed sw/hw pipeline. Each processing step is assigned to a task of the pipeline. Tasks #0 and #2 can be off-loaded to the FPGA since the corresponding hardware modules are available. But inside functions of both tasks could not be pipelined by using *#pragma HLS PIPELINE* because of branching and converging. Tasks #1 and #3 run on the CPU by using the same function in the binary. Two of the four tasks run in parallel since the ARM CPU on Zynq. The Function Off-loader intercepts *cv::cvtColor* as "the head" of a series of functions and off-loads it. For the rest of the functions, Courier-FPGA intercepts and passes them on to maintain the original processing flow by selecting "Passes Through."

## 4.3 Results

The right side of Fig. 4 illustrates the off-loaded result. Courier-FPGA replaced functions and maintained the original flow by selecting "Pass Through." However the process of the built pipeline is the same as the original one, predefined accelerated modules are run on a PL of Zynq.

**Table 1** shows the average processing times when we ran 200 video frames. Courier-FPGA shortened the processing time to 163,510 [$\mu$s] and achieved a 6.1 [fps], or x3.98 speedup compared with the original binary. In Table 1, "Original CPU" indicates the target binary running on the CPU, and "Courier-FPGA" is the final result. AXIvideo2Mat is input to the hardware module via

**Table 1** Processing time comparison of HOG ([$\mu$s]).

| | Original Binary | Courier-FPGA |
|---|---|---|
| Processing Task #0 | | |
| cvtColor | 21,408 | 27,044 |
| Sobel_x | 63,214 | (x5.05) |
| Sobel_y | 51,936 | (on FPGA) |
| Processing Task #1 | | |
| cartToPolar | 208,174 | 187,545 |
| Processing Task #2 | | |
| threshold | 23,995 | |
| convertScaleAbs | 103,947 | 36,524 |
| subtract | 34,192 | (x5.80) |
| threshold | 23,856 | (on FPGA) |
| add | 25,982 | |
| Processing Task #3 | | |
| divide | 46,467 | 53,771 |
| Total (Average) | 650,856 | 163,510 |
| Speed-up | x1.00 | x3.98 |

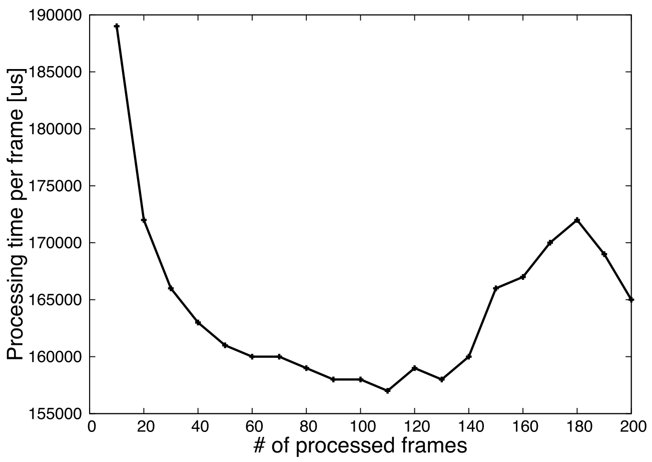


**Fig. 5** Processing time per frame fluctuates a little since Intel TBB's pipeline is based on a thread pool. It works differently from a pure hardware pipeline.

**Table 2** Evaluation of HOG: Frequency, Latency and Exec. time.

| Module | Freq. [MHz] | Latency [clk] | Exec. time [$\mu$s] |
|---|---|---|---|
| Task #0 | 172.1 | 4,654,817 | 27,044 |
| Task #2 | 152.4 | 5,567,778 | 36,524 |

AXI bus and Mat2AXIvideo.

The generated four-stage mixed sw/hw pipeline works well. "Total (Average)" is smaller than the pipeline's Task #1 since TBB searches for and runs an idle task from the thread pool. According to our processing log, Task #0 runs multiple times and stores multiple results while Task #1 runs. Additionally, Task #0 finished the 50th image while Task #1 was processing the 49th image. This pipeline mechanism is different from the ordinary hardware pipeline in which the following stages cannot start until the previous stage has finished. As a result, the average processing time of a single image becomes shorter than the time taken by Task #1. **Figure 5** is a graph showing the relationship between processing time per frame and the number of processed frames. The graph shows 155,000 [$\mu$s] is the lower limit for this task pipeline.

**Tables 2** and **3** show the evaluation of the modules generated for Task #0 and Task #2. The hardware sped up Task #0 by 5.05 times and Task #2 by 5.80 times (this time includes data communications via the AXI Stream bus). In the case of Task #2, there is no Mat2AXIvideo in Table 3 because the AXI4 Stream can be used as a bidirectional port when the bus widths of the input and output are the same. The bus widths of the input/output of Task #2 are 8 bits. On the other hand, those of Task #0 are 32 bits and 8 bits.

**Table 3** Evaluation of HOG: Resource utilization of modules.

| Module | BRAM | DSP48E | FF | LUT |
|---|---|---|---|---|
| Task #0 | | | | |
| Task#0 total | 3(1%) | 9(4%) | 1080(1%) | 1574(2%) |
| AXIvideo2Mat | 0 | 0 | 235 | 279 |
| hls::cvtColor | 0 | 3 | 183 | 154 |
| hls::Sobel | 3 | 6 | 580 | 891 |
| Mat2AXIvideo | 0 | 0 | 44 | 106 |
| Others | 0 | 0 | 38 | 144 |
| Task #2 | | | | |
| Task#2 total | 0(0%) | 14(6%) | 2444(2%) | 4224(8%) |
| AXIvideo2Mat | 0 | 0 | 91 | 126 |
| convert ScaleAbs | 0 | 14 | 2194 | 3733 |
| hls::Threshold | 0 | 0 | 63 | 183 |
| hls::AddWeighted | 0 | 0 | 48 | 91 |
| hls::SubS | 0 | 0 | 48 | 91 |
| Others | 0 | 0 | 73 | 209 |

**Table 4** Processing time comparison of cornerHarris ([$\mu$s]).

| | Original Binary | Courier-FPGA | Running on |
|---|---|---|---|
| cornerHarris | 974.9 | 14.1 | FPGA |
| normalize | 90.0 | 78.5 | CPU |
| convert ScaleAbs | 221.6 | 13.7 | FPGA |
| Total (Average) | 1286.5 | 58.3 | — |
| Speed-up | x1.00 | x22.1 | — |

**Table 5** Processing time comparison of glRotatef ([$\mu$s]).

| | Original Binary | Courier-FPGA | Running on |
|---|---|---|---|
| glLoadIdentity | 18.8 | 17.8 | CPU |
| gluLookAt | 18.1 | 19.0 | CPU |
| glLightfv | 17.8 | 18.1 | CPU |
| glRotatef | 18.4 | 1.9 | FPGA |
| Total (Average) | 73.1 | 56.8 | — |
| Speed-up | x1.00 | x1.29 | — |

## 4.4 Other Case Studies

We conducted other case studies to demonstrate feasibility of Courier-FPGA. Both can be obtained from websites.

### 4.4.1 cornerHarris

cornerHarris_Demo is a sample program of corner detection that is contained in OpenCV (opencv-2.x.y/samples/cpp/tutorial_code/TrackingMotion/cornerHa-rris_Demo.cpp). The binary was mainly composed of three functions listed in **Table 4**. Inputed image size was 1,920 × 1,080. Courier-FPGA built a three-stage pipeline, and x22.1 speed-up was achieved compared with the original binary.

### 4.4.2 glRotatef

hello_world_in_glsl is a simple program of OpenGL and can be downloaded from the website [15]. Four functions listed in **Table 5** are targeted. We implemented a correspinding hardware module of *glRotatef* that performs some single precision floating point matrix calculation [16]. Courier-FPGA built a single-stage pipeline because of the data structure of OpenGL. A 1.29 times speedup was achieved.

## 5. Related Work

There has been an enormous amount of research on hardware-software co-design for reconfigurable systems. Most of them use source code and sophisticated compilers or HLS techniques unlike Courier-FPGA, which only requires binary code running on the host CPU.

Also, there are a number of researches targeting binary acceleration [17], [18], [19], [20], [21]. Most of these researches focus on analyzing instruction-level behavior and translating fine-grained dataflows into hardware circuits. Stitt et al. proposed *Warp Processing*, which takes advantage of the reconfigurability of the FPGA [17], [18]. Their unique points are the original CAD module, which analyzes the code to detect hot-spots, and automatic generation of FPGA circuits. Bispo et al. proposed hardware-based instruction bus profiling to measure the branch frequency of loops (*Megablock*) [19]. Although Megablock can off-load a large block with a number of instructions, it requires special hardware for profiling and off-loading. Nathan et al. presented the *Configurable Compute Array* (CCA) for automatically designing new instruction sets by using an FPGA as a substitute for a series of existing operations on the CPU [20]. They also show how to determine the operations by using both a dynamic profile and static one. Other researches on automatic transformation of assembly language to hardware modules have been done. For example, *eMIPS* [22], *Binary-translation Optimized Architecture* (*BOA*) [23] and *Dynamic Instruction Merging* (DIM) [21]. These studies try to convert the basic blocks in a software binary into a hardware module, and proposed specific means for doing so.

Unlike the above studies, the target of Courier-FPGA is a coarse-grained dataflow. It focuses on generating a task-level pipeline with the cooperation of the host CPU and FPGA considering data transfer between modules. Courier-FPGA assumes that the corresponding HDL description of the target function exists or is easy to be generated with HLS techniques. Thus, Courier-FPGA can be combined with traditional HLS techniques or binary translation techniques which focus on acceleration of individual functions.

## 6. Conclusion

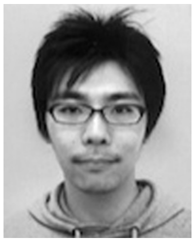
This paper presented *Courier-FPGA*: a new toolchain for application acceleration on a CPU-FPGA platform. The *Backend* of Courier-FPGA builds and deploys a mixed software hardware task pipeline by using the *Pipeline Generator* and *Function Off-loader*. The Pipeline Generator generates software functions and hardware modules. It also makes a pipeline control program by using an Intel TBB in order to run software and hardware tasks in parallel. The Function Off-loader replaces the functions in a target binary with the built pipeline. In the case studies, the running binary of three algorithms were accelerated on the Zynq platform by using Courier-FPGA. As a result, a binary of histogram of gradients (HOG) was sped up 3.98 times. And two other cases were also sped up 1.29 to 22.1 times without user intervention.

In our future work, we will research how to generate a more flexible task pipeline to meet user constraints. For example, resource utilization or power consumption.

**Acknowledgments** The present study is supported in part by the JST/CREST program entitled "Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era" in the research area of "Development of System Software Technologies for post-Peta Scale High Performance Computing."

### References

[1] OpenCV port · Projects · RocketBoards.org, available from ⟨http://www.rocketboards.org/foswiki/Projects/OpenCVPort⟩.
[2] Xilinx Vivado Design Suite User Guide: High-Level Synthesis (UG902), available from ⟨http://www.xilinx.com/support/documentation/sw_manua-ls/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf⟩.
[3] HIPAcc The Heterogeneous Image Processing Acceleration Framework, available from ⟨http://hipacc-lang.org⟩.
[4] Miyajima, T., Thomas, D. and Amano, H.: A Domain Specific Language and Toolchain for OpenCV Runtime Binary Acceleration Using GPU, *2012 3rd International Conference on Networking and Computing* (*ICNC*), pp.175–181 (2012).
[5] Ian Foster: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman Publishing Co., Inc. (1995).
[6] Zynq-7000 Programmable SoCs, available from ⟨www.xilinx.com⟩.
[7] Arria V SoC FPGA Hard Processor System, available from ⟨www.altera.com⟩.
[8] Man page of PTHREADS, available from ⟨http://linuxjm.sourceforge.jp/html/LDP_man-pages/man7/pthreads.7.html⟩.
[9] boost C++ libraries, Chapter 30. Thread 4.1.0 - 1.54.0, available from ⟨http://www.boost.org/doc/libs/1_54_0/doc/html/thread.html⟩.
[10] OpenMP.org, available from ⟨http://openmp.org/wp/⟩.
[11] glibmm, Glib::Thread Class Reference, available from ⟨https://developer.gnome.org/glibmm/2.35/classGli_1_1Thread.html⟩.
[12] Xilinx AXI Reference Guide (UG761), available from ⟨http://www.xilinx.com/support/documentation/ip_docume-ntation/ug761_axi_reference_kguide.pdf⟩.
[13] Navneet Dalal and Bill Triggs: Histograms of Oriented Gradients for Human Detection, *International Conference on Computer Vision & Pattern Recognition* (2005).
[14] OpenCV (Open Source Computer Vision), available from ⟨opencv.willowgarage.com/wiki⟩.
[15] Hello World in GLSL, Lighthouse3d.com, available from ⟨http://www.lighthouse3d.com/tutorials/glsl-tutorial/hello-world-in-glsl⟩.
[16] glRotatef - OpenGL Reference Pages, available from ⟨http://www.opengl.org/sdk/docs/man2/xhtml/glRotate.xml⟩.
[17] Lyseckya, R., Vahida, F. and Tan, S.: A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA Compilation, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.57–62 (2005).
[18] Vahid, F., Stitt, G. and Lysecky, R.: Warp Processing: Dynamic Translation, *Computer*, Vol.41, No.7, pp.40–46 (2008).
[19] Bispo, J., Paulino, N., Cardoso, J.M.P. and Ferreira, J.C.: From Instruction Traces to Specialized Reconfigurable Arrays, *Proc. 2011 International Conference on Reconfigurable Computing and FPGAs, RECONFIG '11* (2011).
[20] Clark, N., Kudlur, M., Park, H., Mahlke, S. and Flautner, K.: Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization, *International Symposium on Microarchitecture*, pp.30–40 (Dec. 2004).
[21] Beck, A.C.S., Rutzig, M.B., Gaydadjiev, G. and Carro, L.: Transparent reconfigurable acceleration for heterogeneous embedded applications, *Proc. Conference on Design, Automation and Test in Europe* (2008).
[22] Pittman, R.N., Lynch, N.L., Forin, R., Pittman, R.N., Lynch, N.L. and Forin, R.: *eMIPS, A Dynamically Extensible Processor*, Microsoft Research Technical Report MSR-TR-2006-143 (2006).
[23] Gschwind, M., Altman, E.R., Sathaye, S., Ledak, P. and Appenzeller, D.: Dynamic and Transparent Binary Translation, *IEEE Computer*, Vol.3 No.33, pp.54–59 (2000).



**Takaaki Miyajima** recieved his B.E. degree from Meiji University, Japan, in 2009. He is currently a Ph.D. candidate at Keio University. His research interests include the areas of design methodology for heterogeneous platform and algorithm implementation.

**David Thomas** received his M.Eng. and Ph.D. degrees in computer science from Imperial College London, in 2001 and 2006, respectively, Since 2010, he has been a Lecturer with the Electrical and Electronic Engineering Department, Imperial College London. His research interests include hardware-accelerated cluster computing, FPGA-based Monte Carlo simulation, algorithms and architectures for random number generation, and financial computing.

**Hideharu Amano** received his Ph.D. degree from Keio University, Japan in 1986. He is now a Professor in the Department of Information and Computer Science, Keio University. His research interests include the areas of parallel architectures and reconfigurable computing.