IPSJ Transactions on System LSI Design Methodology Vol.8 105–115 (Aug. 2015)

[DOI: 10.2197/ipsjtsldm.8.105]

Regular Paper

Courier: A Toolchain for Application Acceleration on Heterogeneous Platforms

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

Received: December 5, 2014, Revised: March 13, 2015,
Accepted: April 29, 2015, Released: August 1, 2015

Abstract: Computationally intensive applications using an open-source library such as OpenCV, BLAS or FFT are widely available on various research or industry applications. Although the optimized code of such libraries has been prepared for an accelerator, off-loading is difficult for non-expert users, especially when only binary of applications can be accessed. This paper presents a new toolchain for application acceleration called *Courier*. It only requires a executable binary of the target application and a corresponding function code for an accelerator. Besides, it doesn't require a source code of the application nor re-compilation of the binary. A work-flow of Courier is a simple and intended for non-expert users. It extracts runtime information from running binary, generates task graph, and then replaces the original function with a corresponding accelerator function. Many steps along with the application acceleration process are automatically executed. The users can refer to the acceleration result and modify the task graph if needed. In our case studies, Courier was used for acceleration of three applications; image processing, matrix multiplication and spectrum analysis. Functions are off-loaded to a GPU without any modification to the original source code. Applications are sped up 8.89, 8.16 and 1.23 times, respectively.

Keywords: algorithm implementation, heterogeneous platform, automation tool

1. Introduction

For expert programmers, performance of computationally intensive applications can relatively easy to be improved by off-loading time consuming parts to accelerators like GPUs or FPGAs. On the contrary, users of legacy or public applications do not have enough knowledge on their executing processing flow, and often they do not have the source code itself. For such users, performance improvement with accelerators have been almost impossible. However, recently, a lot of applications use widespread function libraries like OpenCV, BLAS or FFT, and optimized off-the-shelf code of such functions are already available for popular accelerators [1]. If we understand the flow of the running binary and find the parts which can be accelerated, we can off-load the binary by replacing the parts with the corresponding functions of the accelerator. Here, we propose a new toolchain for application acceleration called *Courier*. Courier automatically analyses specific functions and data in a running binary and replaces functions with corresponding accelerator functions if possible.

The contributions of the paper are as follows:

- Introducing a new application acceleration work-flow, *Courier*, which does not require original source code, manual tweaks, or re-compilation of the target binary, without user intervention. The user just has to refer to the result and modify off-load parts if needed.
- We propose an automatic processing flow graph generation method of analyzable functions from a running binary. The method includes tracing sub-programs to analyze functions and a heuristic approach to detect causality.
- We propose an automatic off-loading method of functions in the binary. If functions are analyzed by the above mentioned method and corresponding functions are ready for the accelerator, functions are off-load automatically. The method also reduces the number of data transfer along with off-loading, and maintains an original processing flow before and after off-loading.
- We show practical case studies: a HOG feature detection with OpenCV, a matrix multiplication using BLAS and a power spectrum density estimation using FFT. These were sped up 8.89, 8.16 and 1.23 times by using the existing GPU functions.

This paper is organized as follows. In Section 2, we present Courier, including its features designed for detecting a processing flow and function off-loading. Then, we describes technical details of our function off-loading mechanism. Section 4 gives three case studies, showing the capability of Courier. We discuss our toolchain and related work in Section 5. Finally, we conclude the paper.

2. Courier: A Toolchain for Application Acceleration

2.1 Target Users and Applications

Most of researches on developing work-flow of off-loading fo-

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

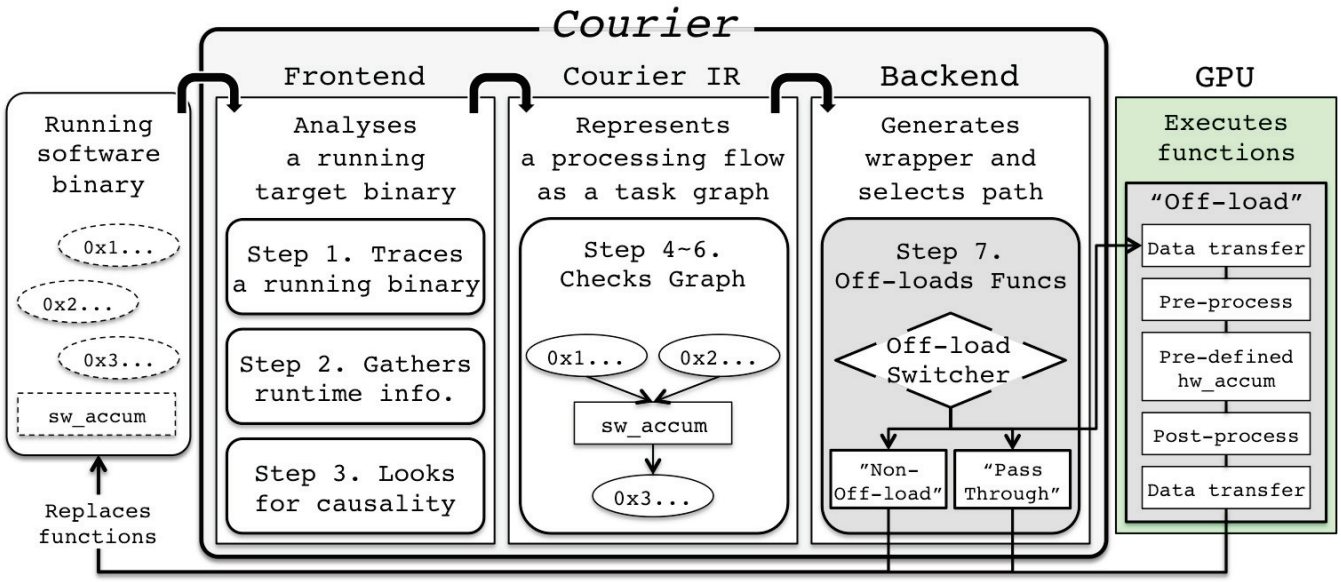


Fig. 1 An overview and work step of the Courier: *Frontend* traces running binary (1, 2) and detects causality (3) and then generates an *Intermediate Representation* (4) and a task graph (5). Users modify off-load parts and changes IR if needed (6). Finally, *Function Off-loader* replaces function and off-loads to accelerator (7).

cus on a programmer who knows of the source code and wants to improve its performance. For developing the code of the accelerator which works the same function of the target code with much more performance, various types of tools have been proposed. They help the analysis of the source code [2], accelerator management [3], [4], and accelerator kernel implementation [5], [6], [7], [8]. Unlike them, we do not intend to generate the accelerator code itself. We assume that the target application program uses a common library like OpenCV, BLAS or FFT, and the corresponding library code of the accelerator is already available. Instead, our target users are not needed to have the source code of acceleration target. Our toolchain extracts the call flow of functions, and find the part which can be off-loaded to the accelerator during execution of the binary code. Current version of the toolchain cannot do anything if the target binary does not include corresponding accelerator functions. Even with this limitation, the proposed tool can help many legacy code users who are out of the target of the conventional work-flow.

2.2 Overview of Courier

Figure 1 shows the overview of Courier. It is consisting of *Frontend* (*Runtime Analyzer*) and *Backend* (*Function Off-loader*). Courier Intermediate Representation (IR) is a bridge between them.

Example of the processing step is illustrated in Fig. 1 and caption of the figure describes the detailed work step. Running software binary contains a function called "accum," which obtain two input data (0x1 and 0x2) and produces an output data (0x3). Courier traces the binary and detects "accum." Then Courier replaces the function with corresponding accelerator function "acc_accum." We are go into more detail in the next sub sections.

2.3 Frontend (Runtime Analyzer)

Frontend is consisting of three main steps so as to detect a raw processing flow. We used dynamic program analysis and a heuristic approach to detect the flow. Users simply start their application as usual, and Courier performs a data sampling process called a "profile run." Each step to analyze running binary works as follows during profile run.

Step 1. Frontend traces functions in the running binary by using pre-defined tracing sub-programs,
Step 2. gathers runtime information, during execution,
Step 3. and looks for causal function call and input/output data.

The purpose of extracting a function-level processing flow is NOT to translate assembly automatically into accelerator kernel, like the fine grained dataflow [8]. We intend to understand processing flow and find parts that can be off-loaded on a current heterogeneous platform.

2.3.1 Tracing a Running Binary

In Step 1 we perform dynamic program analysis by using tracing sub-program for each function in order to obtain runtime information from a target binary. In Step 2 Frontend gathers these information. The information includes absolute time of entry and exit, thread id, call depth, function name, and raw argument value (not just the memory address). At runtime, all dynamic pointers/aliases (e.g., *int** in C++) are resolved, so the raw value is available. Frontend covers a super-set of manual profiling and can gather more information than the other research [8], [9], [10]. In this paper, we call the value that is actually processed at runtime the "raw value."

Tracing sub-programs is based on Intel Pin, a framework for dynamic program analysis [11]. Pin is commonly used for kernel-level profile of the target program, but we use it to detect a function-level processing flow. Many libraries such as OpenCV, Basic Linear Algebra Subprograms (BLAS), and Fast Fourier Transform (FFT) can be simultaneously analyzed by preparing specific tracing sub-programs. Note that Courier uses conven-

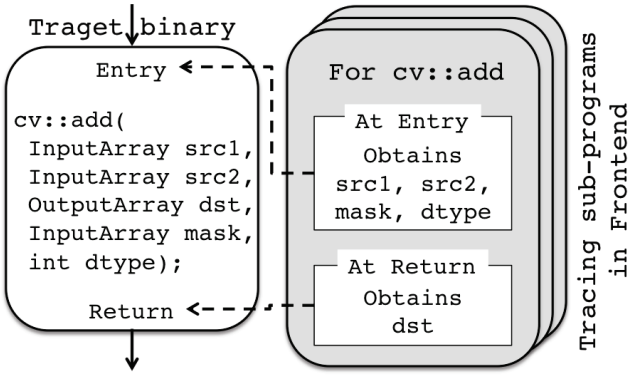


Fig. 2 Tracing sub-programs for specific functions in Frontend access designated arguments and obtain raw values at entry and return points of the functions.

tional tools only in this step. To analyze a specific function, some information of a target library must be known in advance. Specifically, information of a structure of data type, functions name and role of each argument are required. Tracing sub-program is a separated from Courier implementation so as to improve applicability to support new libraries. By adding new tracing sub-program for a specific library, Courier can trace the library.

Figure 2 shows a target function cv::add on the left and a tracing sub-program for it on the right. Note that "Entry" and "Return" are points of function where control enters or exits each function region. During the profile run, Frontend accesses a raw address of 1st and 2nd arguments and obtains a raw value (input data) at the entry point of cv::add. 4th and 5th arguments are also accessed at that time. At a return point, Frontend accesses 3rd argument and obtains a raw value (output data). Absolute times are also recorded. Listing 1 is gathered runtime information of cv::add. It's just an enumeration of the information and hidden from the users.

2.3.2 Looking for Causality

A heuristic approach is used to look for causal processing flow between functions and input/output data dependencies in Step 3. For example, assume that a function, named *cv::divide* which has an argument that contains input data, is found after cv::add. If an output data of cv::add and an input data of cv::divide are the same, Frontend guesses the causality by which these two functions are connected by the data and detects processing flow like "cv::add" → "cv::divide." Raw value is typically non-identical, e.g. images are less likely than chance to match. Even if some unrelated raw values are the same or functions run in parallel, Frontend detects causality by referring to time, thread id, or call depth.

2.4 Courier Intermediate Representation (IR)

Courier IR is an intermediate representation that bridges Frontend and Backend. It can be used to modify or designate parts to off-load if the users don't satisfy Courier's automatic off-load. The three main steps of Courier IR are as follows.

Step 4. Courier generates an IR corresponding to the detected processing flow,
Step 5. generates a *task graph*, and
Step 6. the users modify or designate off-load parts if needed.

All information gathered from Frontend such as Listing 1 is

```
[ENTRY]:
cv::add(cv::_InputArray const&,cv::_InputArray const&,
cv::_OutputArray const&, cv::_InputArray const&, int)
[TIME]:
33337
[ARGs]:
0x7fffd6b4cfd0, 0x7fffd6b4cfb0,
0x7fffd6b4cf90, 0x7fbdc9325540, 0xffffffff
[IMG]:
0x42ff4005, 1920, 1080, 0x26f4f30
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...
[IMG]:
0x42ff4005, 1920, 1080, 0x2613f00
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...
[RETURN]:
cv::add(cv::_InputArray const&,cv::_InputArray const&,
cv::_OutputArray const&, cv::_InputArray const&, int)
[TIME]:
39939
[IMG]:
0x42ff4005, 1920, 1080, 0x27d5f60
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...
```

Listing 1 Gathered runtime information of cv::add.

```
img3 = cv::add(img1, img2);
```

Listing 2 Courier IR description of cv::add.

translated into a more user-friendly description as shown in Listing 2. Its structure is simple and device independent and shows the inferred function-data causality. By just lining up functions and data in the order of processing flow. IR is converted into graph representation called *Task Graph*. Task graph is a kind of weighted directed acyclic graph and includes order of function call, their input/output, and some raw values. At present, Courier IR is manually translated from the detected processing flow. Some special functions are provided to designate the off-load functions and maintain original dataflow. *cpu2acc* function designates off-load function and input/output data from CPU to the accelerator, and *acc2cpu* function does the same in the opposite direction. *volatileInput/Output* functions are automatically called to notify the users that they cannot change or delete certain nodes, due to the need to protect the overall inputs and outputs of the task graph. We show an example in Section 4.1 with an example of HOG feature detection.

2.5 Backend (Function Off-loader)

Backend is designed for automatic off-load without user intervention, and a main step is as follows.

Step 7. The *Function Off-loader* selects a path and replaces functions with corresponding accelerated functions within the designated parts.

Backend first searches for "safely off-loadable" parts, where input and output data are both traced, data conversion is feasible, and a corresponding accelerated function is available. For such parts, Backend automatically off-loads them by using *Function Off-loader* in default mode. Note that we do not attempt any sort of automatic binary translation. We explain the details in Section 3 with an example of OpenCV.

2.6 Applicability of Courier

When the users want to support a new library, they should in-

troduce a new "add-on" for Courier. Add-on is a supplementary component that improves capability without changing the main application. Add-on for Courier includes a tracing sub-program, a data transfer function, a corresponding accelerator function and a correspondence relationship of accelerator functions. By using this add-on mechanism, Courier can easily support new library without developing a new version of Courier.

When the corresponding functions are ready in the accelerator, any types of function calls in the target binary can be off-loaded. In the case of multiple function calls that appear continuously, Courier off-loads all of them at a time and the performance can be much improved by using Function Off-loader even if additional statements are existing before and after functions. On the other hand, if there are additional statements between functions, they are executed in the host processor and each function is off-loaded independently. In this case, each function call requires the data transfer between the host processor and the accelerator, and the performance improvement might be degraded if the granularity of the function is not large compared with the data transfer time. In the current version of Courier, the users eventually select whether the function is off-loaded or not by using the Off-load Switcher. Additionally, Courier cannot off-load function calls of non-supported library. They are executed on the host processor as the same way as the additional statements.

3. Function Off-Loading System

Function Off-loader and *Off-load Switcher* in the Backend are core features of application acceleration. It automatically generates a function wrapper to replace the original function designated by IR's *cpu2acc* and *acc2cpu*. The wrapper contains code of the pre-defined corresponding accelerator function, a pre/post-processing and the data transfer. The Backend creates a shared object from the code. Function off-loading system behaves as follows. At start-up, Courier stops the running binary, and then Function Off-loader intercepts (hooks) designated functions. It then replaces original functions with the wrapper that executes the accelerator functions while maintaining processing flow or reducing the data transfer by using *Function Switcher*. Finally, Courier re-starts the binary. This process does not require any user intervention. The corresponding accelerator functions must be available beforehand, so we use OpenCV's GPU functions, cuBLAS and cuFFT in Section 4.

3.1 Dynamic Linking and Its Problems

We used dynamic linking mechanism on Linux environment for the Function Off-loader as a basis. This mechanism adjusts the runtime linking process by forcibly loading and linking software libraries. Source-code tweaks or re-compilations of target binary are NOT required. Function Off-loader uses it to replace original functions in the binary with wrappers. Wrappers needs to be compiled before the deploy. Although this technique is known as DLL hijacking or DLL injection, here, the purpose is off-loading and some problems are occurred.

There are three main problems that occur if we just use DLL injection for the function off-loading. The first is unconditional off-loading (**Fig. 3**, UNCOND-OFF), the second is a restriction

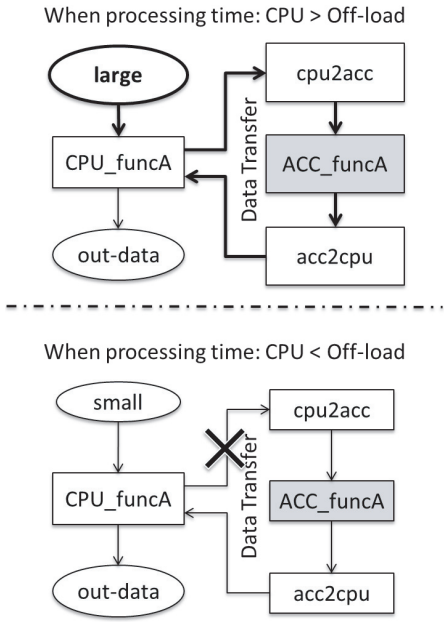


Fig. 3 UNCOND-OFF: Typical dynamic linking replaces all the same name functions in the target binary.

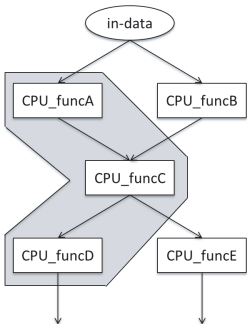


Fig. 4 SAME-INOUT: Corresponding function must have the same number of input output.

of the number of inputs out of substitute function (**Fig. 4**, SAME-INOUT), and the third is redundant data transfer when a series of functions is off-loaded (**Fig. 5**, RDNT-TXRX). In the figures, ellipse nodes and rectangle nodes represent data and functions, respectively. The UNCOND-OFF problem is caused by DLL injection, since DLL injection replaces all the same name functions in a target binary unconditionally. This is not suitable for off-loading, since processing time on heterogeneous platform usually depends on data transfer overhead. In the case of Fig. 3, assume that there is two "CPU_funcA" functions in binary and both are replaced. The data size of "srcImg0" including communication overhead is large enough to off-load. On the other hand, "srcImg2" is too small for off-loading, since the total processing time (communication overhead + execution time on accelerator) is longer than processing time on CPU. Thus, only intended functions should be off-loaded. The SAME-INOUT problem forces Courier to use a function that has the same number of inputs/outputs as that of the original function. Some opportunities for off-loading are lost by this restriction. We are researching a solution to solve this problem, but this is future work. RDNT-TXRX problem arises when series of functions are off-loaded, data transfer happens along with each function, and performance degrades. To deal with UNCOND-OFF and RDNT-TXRX, we introduced

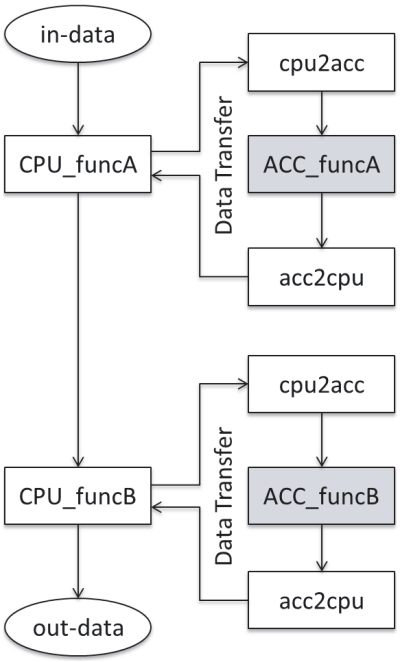


Fig. 5 RDNT-TXRX: Redundant data transfer happens when a series of functions are off-loaded.

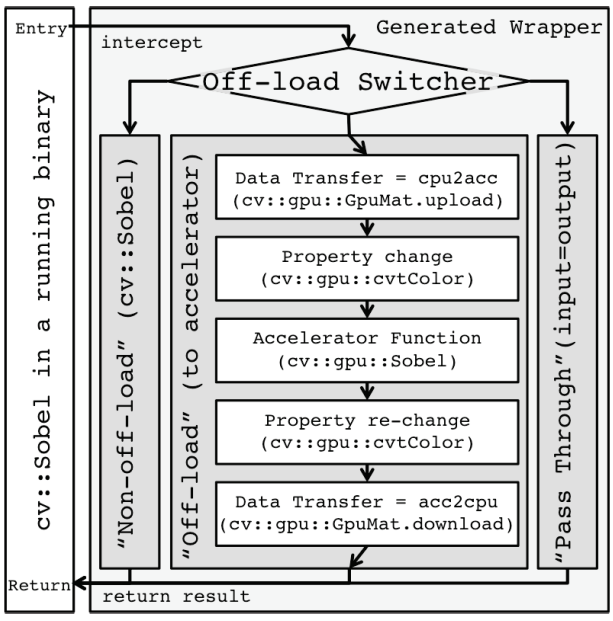


Fig. 6 Function Off-loader and generated wrapper for OpenCV: One of three paths is selected: *"Non-off-load," "off-load"* and *"Pass Through."*

Function Off-loader and *Off-load Switcher*.

3.2 Mechanism of Function Off-loader

Function Off-loader generates a wrapping code around accelerator functions code and solves the above mentioned two problems. To generate an appropriate code, it has a table which contains a correspondence relationship between software functions and accelerated functions, code of a needed pre/post-processing and the data transfer. **Figure 6** shows an example wrapper for the *cv::Sobel* function in OpenCV. In the figure, the wrapper introduces a data transfer function, *cv::gpu::Sobel* and *cv::gpu::cvtColor*. *cv::gpu::Sobel* is an corresponding accelerator function for cv::Sobel. *cv::gpu::cvtColor* is a pre/post-process that adjusts image properties between cv::gpu::Sobel and cv::Sobel. The additional overhead of the wrapper is transferring image and property conversion. It depends on the image size.

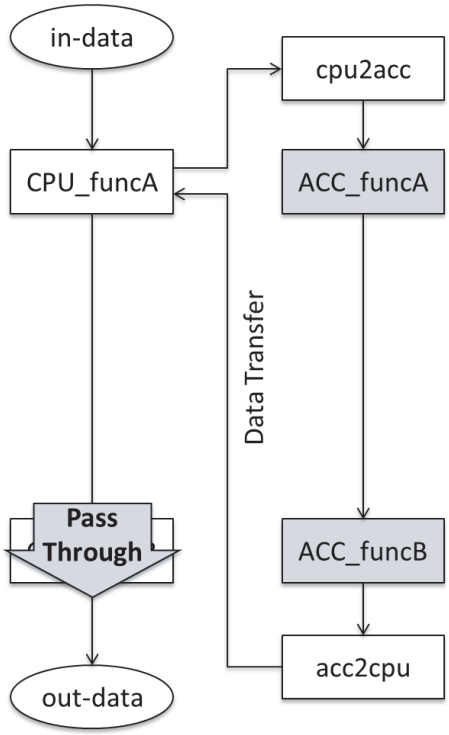


Fig. 7 *Function Off-loader* reduces the # of data transfer and maintains the original processing flow.

To deal with the above described UNCOND-OFF and RDNT-TXRX problems, we introduce *Off-load switcher* to the wrapper. This switcher provides one of three possible paths for a function: *non-off-load*, *off-load* and *pass through*. The path is selected by Function Off-loader and determined from arguments of function or function ID that is contained in Courier IR. This uses dlsym and dlopen [12], which are APIs for dynamic loading in Linux. In Fig. 6, *Off-load switcher* is shown at the top, and the three paths work as follows.

- *Non-off-load* keeps the function the original, so the function runs on CPU.
- *off-load* replaces the designated function with corresponding accelerator function. Some pre/post-processing is also added.
- *Pass Through* assigns the input data to the output data so as to skip the function in binary.

The UNCOND-OFF is solved by executing original function in a non-off-load path, and the RDNT-TXRX is solved by the following method. Function Off-loader replaces "the head" of a series of functions and runs all functions in it. Figure 5 and **Fig. 7** illustrate before and after suppression, respectively. Moreover, to maintain original processing flow, successive functions must be skipped in the original binary running on CPU. Otherwise they are applied twice in the off-loaded function and original binary. Thus, our Function Off-loader replaces and skips them by using *Pass Through*. The off-load switcher is controlled by gathered information from Frontend, such as function name, argument value,

```
1  #include "opencv2/imgproc/imgproc.hp>"
2  int hog (rtnMagnitude, rtnHistogram){
3   while(true){
4  // Step 1) Compute gradient and magnitude
5    // input and gray scale
6    cv::VideoCapture cap >> frame;
7    cv::cvtColor(frame, yuv, "CV_RGB2YCrCb");
8    cv::split(yuv, graySc);
9    // x/y-axis Sobel filter
10   cv::Sobel(graySc, xsobel, "X-axis");
11   // image duplication via memcopy
12   graySc.copyTo(copyTo_dst);
13   cv::Sobel(copyTo_dst, ysobel, "Y-axis");
14   // Calculate gradient and magnitude
15   cv::cartToPolar(xsobel,ysobel,gradient,
          magnitude);
16 // Step 2) Gradient adjustment
17   // Adjust the value within 0 to 180 degrees
18   cv::threshold(gradient,  180up, "<180");
19   cv::convertScaleAbs(180up, 180up_res);
20   cv::subtract(180up_res, 180matrix, sub_res);
21   cv::threshold(gradient, 180low, "180<");
22 // Step 3) Create histogram
23   cv::add(sub_res, 180low, add_res);
24   cv::divide(add_res, div_in, histogram); }}
```

Listing 3 Pseudo code of HOG feature detection in the target running binary.

```
1  void hog.o_main(void){
2    // Original Input/Output, unchangeable
3    volatileInput(frame);
4    volatileInput(180matrix);
5    volatileInput(div_in);
6    volatileOutput(magnitude);
7    volatileOutput(histogram);
8
9    frame = cv::VideoCapture();
10   yuv = cv::cvtColor(frame);
11   graySc = cv::split(yuv);
12   xsobel = cv::Sobel(graySc, "X-axis");
13   copyTo_dst  = cv::Mat::copyTo(graySc);
14   ysobel = cv::Sobel(copyTo_dst, "Y-axis");
15   {gradient, magnitude} // generates two results
16           = cv::cartToPolar(xsobel,ysobel);
17
18   180up = cv::threshold(gradient, "<180");
19   180up_res = cv::convertScaleAbs(180up);
20   sub_res = cv::subtract(180matrix, 180up_res);
21   180low = cv::threshold(gradient, "180<");
22
23   add_res = cv::add(sub_res, 180low);
24   histogram = cv::divide(add_res, div_in);}
```

Listing 4 Generated IR description of the target running binary.

and data size. Note that we don't use execution time to control it currently.

4. Case Study

In this section, we illustrate our work-flow by using three case studies: 1) a HOG feature detection algorithm using OpenCV, 2) a double precision general matrix multiplication using Basic Linear Algebra Subprograms (BLAS), and 3) a power spectrum density estimation using fast fourirer transform (FFT). Binaries are accelerated in a CPU-GPU environment by using Courier. Experimental conditions are as follows: Host OS is Fedora 20 64 bit (kernel 3.14.3), CPU is Intel Core i7-3770K 3.5 GHz, and the accelerator is NVIDIA GeForce GTX670 with PCIe Gen.3. Binaries are compiled with GCC ver.4.7.

4.1 Histogram of Oriented Gradients (HOG)

HOG is a widely used algorithm for feature detection, such as face recognition [13]. HOG application includes three main features that are commonly seen in computer vision applications: OpenCV C++ API functions, diverging/converging flow, and image duplication. The processing flow in running binary is consisting of the following three main steps. Step 1) Compute gradient and magnitude, Step 2) Gradient adjustment, and Step 3) Create histogram.

4.1.1 Acceleration Work-flow of Courier

I. Analyzing running binary

After user designates the target binary, Frontend analyses running binary, then detects processing flow and IR. This step corresponds to Step 1–3 in Fig. 1. By tracing sub-programs the following information are extracted:

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

II. IR Description

The Courier IR description as shown in List 4 is automatically generated. Users modify this to change processing flow if needed (Step 6 in the Fig. 1). At line 9, the *cv::VideoCapture* function is used to provide input images for the processing flow. Images are taken as an argument of *volatileInput* at line 3 to 7, and so it is protected from modification since this is the very first data. Two *cv::Sobel* with different arguments are at lines 12 and 14. Such differences can be detected by a dynamic program profile on Frontend.

III. Generating a task graph

After the profile runs, a task graph of the binary is automatically generated, which is shown on the left of **Fig. 8**. User can refer the graph and decide off-load and non-off-load parts if needed (Step 6 in Fig. 1). The graph is identical to the previously described processing flow. Rectangle nodes and ellipse nodes represent functions and original input/output data, respectively. Edges represent intermediate data. The thickness of the edge also reflects the size of data. Processing times are displayed in the second row of the rectangle node. Nodes are aligned in chronological order. According to the graph, each image is processed in 77,923 [μs] in total.

Rectangle nodes of various sizes allow the user to easily recognize that large nodes (e.g., *cv::convertScaleAbs* or *cv::divide*) occupy a large fraction of total processing time. Two kinds of thickness of edges can be seen in the figure since *cv::Sobel* and *cv::convertScaleAbs* functions change the number of bit-depth of their inputs. The data size of thicker edges is 7.91 Mbit ($1{,}920 \times 1{,}080 \times 32$ bit $\times$ 1-channel) and 1.98 Mbit (the same property, but 8 bit), respectively. Dynamic program analysis on Frontend correctly extracts the runtime information.

The graph also illustrates that this binary includes typical branching and converging, for example both *cv::Sobel* operators use the same image as an input, and these output images become inputs of *cv::cartToPolar*. Furthermore, the vertical relative offsets (separated by dash lines) illustrate sequential execu-

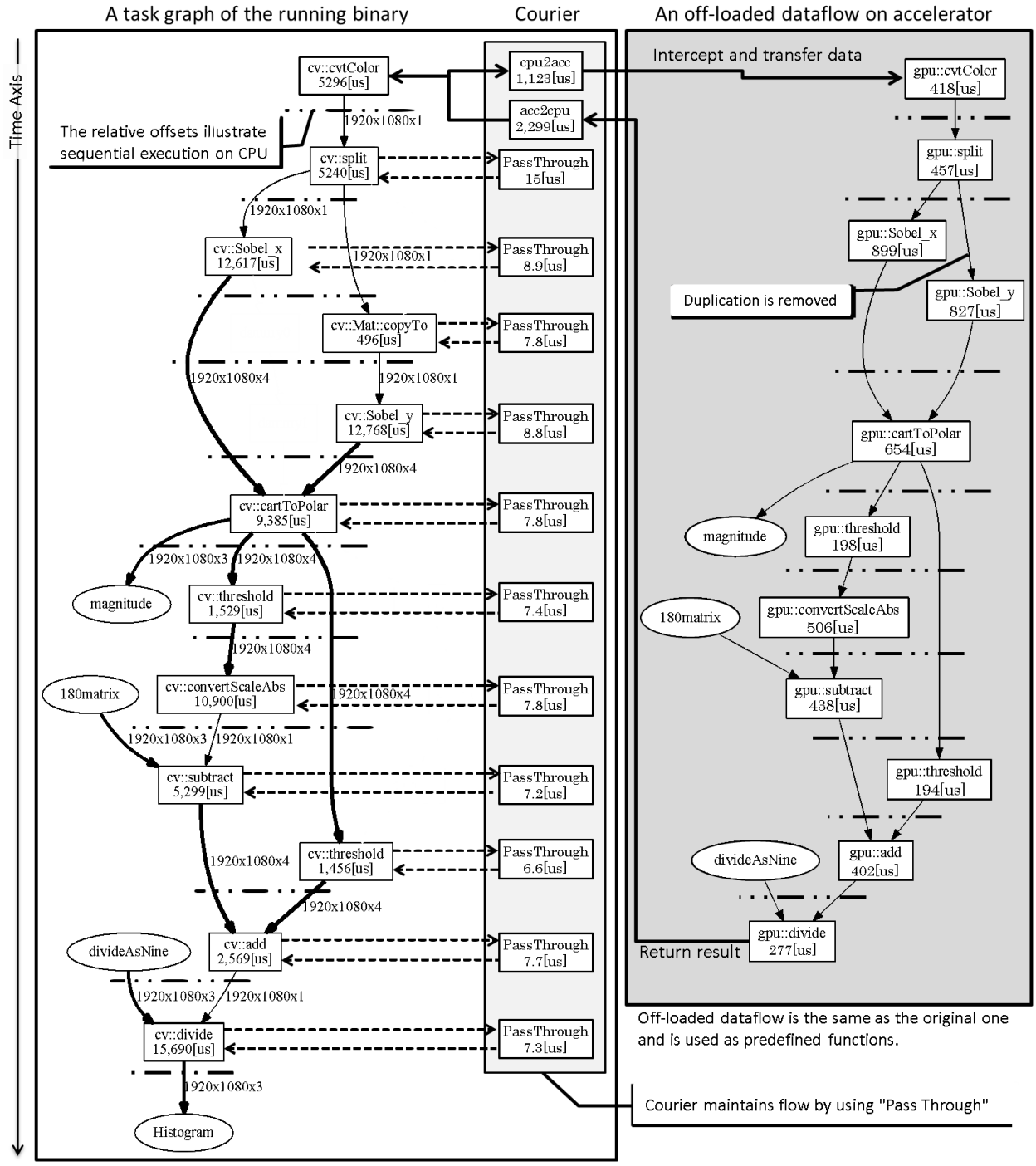


Fig. 8 Generated task graph from running binary of HOG (left) and off-loaded functions (right) with notations. Function Off-loader generates the wrapper for the functions within *cpu2acc* and *acc2cpu*. It also selects the path of "Off-load" and maintains the processing flow by using "Pass Through."

tion, which is an opportunity to exploit function level parallelism. Additionally, *cv::Sobel_y* creates a copy of input images which seems to be unnecessary. After modifying the processing flow with IR, such redundant can be deleted.

IV. Acceleration by Courier

In this step (Step 7 in the Fig. 1), Courier searches for "safely off-loadable" parts, and founds that all of the functions are candidate. Courier automatically off-loads it by using Function Off-loader and existing corresponding library. Finally, Courier updates the IR and introduces the following new lines at the last of List 4:

```
25 // Off-load from cv::cvtColor to cv::divide
26 cpu2acc(cv::cvtColor, MOVE, gpu0);
27 acc2cpu(cv::divide, MOVE, gpu0);
```

Listing 5 Additional statements to off-load functions.

Courier also selects the "Pass Through" pass in Function Off-loader to reduce the number of data transfer and maintain original processing flow. (deal with a RDNT-RXTX problem which is shown in Fig. 7.) For the first purpose, Function Off-loader in-

Table 1 Processing time comparison of HOG ([μs]).

	Original on CPU	Off-loaded functions	Courier's result	Manual imple.
Processing Step1				
cpu2acc	—	1,123	—	1,109
cvtColor	5,296	418	8,690	418
split	5,240	457	15	442
Sobel_x	12,617	899	8.9	956
copyTo	469	—	7.8	—
Sobel_y	12,769	827	8.8	873
Processing Step2				
cartToPolar	9,385	654	7.8	735
threshold	1,529	198	7.4	195
convert ScaleAbs	10,900	506	7.8	607
subtract	5,299	438	7.2	495
threshold	1,456	194	6.6	204
Processing Step3				
add	2,569	402	7.7	408
divide	15,690	277	7.3	284
acc2cpu	—	2,297	—	724
Total	77,923	8,690	8,766	7,450
Speed-up	x1.00	x8.96	x8.89	x10.46

tercepts *cv::cvtColor* as "the head" of a series of functions and "off-loads," and then all functions are run on GPU. For the rest of functions, Courier intercepts and "Passes Through" even if a function doesn't off-loaded.

If we use ordinary DLL injection, nine data transfers happen. Nine data transfers (two round trips for small images, and seven for large ones: $447 \times 2 + 3,176 \times 7 = 23,106$ [μs]) were reduced to one (send small image and send back large one: $1,109 + 724 = 1,833$ [μs]). Data transfer time is reduced to less than 10%, and RDNT-RXTX problem is solved.

Additional tweaks can be performed by the user. According to Fig. 8, *cv::Mat::copyTo* seems a redundancy. Therefore, this copying function can be deleted as below by the user, and Function Off-loader replaces this function with "Pass Through." Because this deletion may affect the processing flow, Courier does not automatically performs it. For such "unsafe" dataflow modification, Courier makes it the responsibility of the user to check whether the final result is the same as the original one or not.

```
13 //  copyTo_dst  = cv::Mat::copyTo(graySc);
14   ysobel = cv::Sobel(graySc, "Y-axis");
```

Listing 6 Delete redundant copyTo.

V. Results

The right side of Fig. 8 shows the off-loaded result. Courier replaces designated functions and maintains the original processing flow by selecting "Pass Through." On the GPU side, off-loaded version is the same as the original one and predefined accelerated functions with wrapper are used. "copyTo" does not run on GPU anymore, and is "Passed Through" in the binary. That is, "copyTo" is deleted.

Table 1 shows processing times. Courier shortened the processing time to 8,766 [μs] and sped up x8.89 compared with the original binary. In Table 1, "Original on CPU" shows the target binary runs on CPU, "Off-loaded functions" is the processing time of each function in off-loaded parts. "Courier's result" is the final result including the overhead of "Pass Through." Note that the processing time of cvtColor is equal to "Off-loaded functions." It shows that cvtColor is replaced by Courier and all functions are executed in here. Additionally, processing time of acc2cpu in "Off-loaded functions" is longer than that of "Manual imple." because current acc2cpu for OpenCV includes additional data copy. The data copy is required in order to assign the result data forcibly from GPU to the binary. "Manual imple." is a manually implemented GPU version of the original application. The difference between "Courier's result" and "Manual imple." arises from *cpu2acc* and *acc2cpu*. Both commands include data type conversion along with data transfer from GPU to CPU. Additionally, the number of data transfer of "Courier's result" is the same as that of "Manual imple." since Function Off-loader reduces the redundant data transfer.

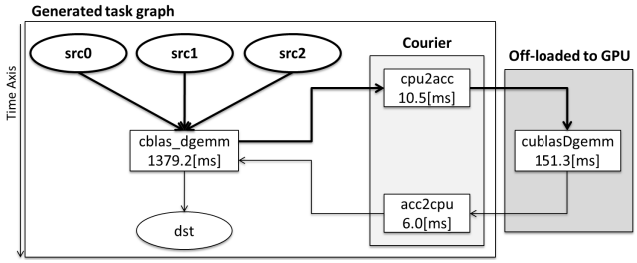


Fig. 9 Generated task graph from dgemm binary.

We also measured overhead of Function Off-loader for OpenCV. We subtract the processing time of "Not-off-load" from ordinary run to measure an overhead of wrapper function and dynamic linking. It is around 150 [μs] for each and is attributed to OpenCV data type conversion and function pointer replacing. The former one is less than 20 [μs] and the latter one is around 130 [μs]. For the "Pass Through," the overhead is around 7 [μs] for each.

4.2 General Matrix Multiplication (GEMM)

General matrix multiplication (gemm) performs the following equation $C = \alpha AB + \beta C$, and very widely used in computational science. *dgemm* is a double precision version of gemm. We prepared binary that only had *dgemm* function of BLAS, a widely used for common linear algebra operations. Binary includes *cblas_dgemm* in ATLAS 3.8.4 [14] for CPU, and Courier uses *cublasDgemm* in cuBLAS [15] in CUDA 6.0 for GPU.

I. Analyzing running binary

We prepared a tracing sub-program for BLAS and it extracts the following information:

- BLAS API function name with arguments,
- function start/end absolute time,
- # of input/output of function, and
- raw value and size of input/output matrix

II. IR Description

The following IR description shows that all the data are protected from modification via *volatileInput/Output*.

III. Generating a task graph

After the profile run, a task graph is generated which is on the left of **Fig. 9**. According to the graph, *cblas_dgemm* is a ternary function, which obtains three data ("src0," "src1" and "src2") and

```
1 void dgemm_cblas.o_main(void){
2 volatileInput(src0); volatileInput(src1);
3 volatileInput(src2); volatileOutput(dst);
4     dst = cblas_dgemm(src0, src1, src2); }
```

Listing 7 dgemm processing flow in Courier IR.

Table 2 Processing time comparision of gemm ([ms]).

	Original on CPU	Off-loaded functions	Courier's result	Manual imple.
cpu2acc	—	10.5	—	10.4
dgemm	1,379.2	151.3	168.7	150.6
acc2cpu	—	6.0	—	5.9
Total	1,379.2	167.8	168.7	166.9
Speed-up	x1.00	x8.16	x8.16	x8.26

produces result data "dst." The matrices are all 2,048×2,048, and consequently all the ellipse nodes are the same size. CPU takes 1,379 [*ms*] to process each matrices.

IV. Acceleration with Courier

Courier founds *cblas_dgemm* is "safely off-loadable" because all the input/output data are extracted, and an corresponding accelerator function *cublasDgemm* is available. Then *cblas_dgemm* is automatically off-loaded. Function Off-loader generates a wrapper which reserves memory and transfers matrices to one GPU memory with the use of *cudaMalloc* and *cublasSet/GetMatrix*, respectively. Here no property change is required. For the IR description, Courier automatically introduces the *cpu2acc* and *acc2cpu* around the *cblas_dgemm*. In this case, all the acceleration processes are done by Courier, and there is no need to do by the user.

V. Results

The right of Fig. 9 shows the off-loaded flow. Processing time is shortened to 151.3 [ms] by *cublasDgemm* on GPU. Including the data transfer and conversion time, total processing was sped up x8.16. **Table 2** shows a final result. Courier achieved almost the same speed up ratio as manual GPU implementation. This is because Function Off-loader just performs the same thing, memory reservation and data transfer, with manual acceleration.

The overhead of Function Off-loader for BLAS is 0.08 [μs]. BLAS has much smaller overhead than that of OpenCV since it does not require data type conversion in this case. Consequently, overhead of Function Off-loader depends on the target function.

4.3 Power Spectral Density Estimation (PSD)

GNU Octave [16] is an open-source software for numerical computations and mostly compatible with MATLAB. Octave accepts user script file for execution. We downloaded a script file that performs power spectral density (PSD) estimation from the website [17]. It includes fast fourier transform (FFT) and other processing. FFT is a widely used routine for numerical analysis. Octave performs FFT by using fftw library [18] on CPU in default. Courier replaced fftw with cuFFT and off-loaded FFT to GPU. GNU Octave 3.6.4 (with fftw 3.4.4) and cuFFT [19] in CUDA 6.0 were used.

I. Analyzing running binary

We prepared a tracing sub-program for FFT in Octave and it extracts the same type of information as for BLAS.

```
1 void octave.o_main(void){
2 volatileInput(src0); volatileInput(src1);
3 volatileOutput(dst);
4     dst = fftw_execute_dft_r2d(src0, src1); }
```

Listing 8 Detected processing flow in Courier IR.

Table 3 Processing time comparision of PSD ([ms]).

	Original on CPU	Off-loaded functions	Courier's result	Manual imple.
cpu2acc	—	0	-	-
fftw_execute_dft_r2c	449.0	99.2	99.2	99.2
acc2cpu	—	0	-	-
Other funcs	821.2	-	868.8	868.8
Total	1,270.2	-	99.2	99.2
Speed-up	x1.00	-	x1.23	x1.23

II. IR Description

IR description is almost the same as BLAS case study. Octave includes many other processes, but Courier cannot analyze them since the current tracing sub-program doesn't support them. By adding information of other functions, an applicability will be improved.

III. Generating a task graph

After the profile run, a task graph was generated. The graph is almost the same as that of BLAS case study and shows that *fftw_execute_dft_r2d* performs actual FFT in fftw library. It performs 16,777,216 points FFT and takes 449.0 [ms] on CPU. Entire processing time of the PSD script is 1,270 [ms].

IV. Acceleration with Courier

Courier founds *fftw_execute_dft_r2c* is "safely ff-loadable" and a corresponding accelerator function (fftw-compatible cuFFT) is available. Function Off-loader generates a wrapper just includes the real-number input DFT function of cuFFT. IR description was also changed just like BLAS case.

V. Results

Processing time of FFT was shortened to 99.2 [ms]. Including the data transfer and conversion time, entire processing time became 968 [ms]. **Table 3** shows a final result. Note that "Total" is entire processing time of the PSD script which includes FFT and other processes. Speed up ratio was the same as manual GPU implementation since cuFFT's FFT function includes pre/post-process. The overhead along with off-load is included in the result and *cpu2acc/acc2cpu* is zero. In this case, wrapper doesn't do any additional processing.

5. Related Work

5.1 Toolchains for Supporting Off-loading

There is a significant amount of existing researches on automatic off-loading systems [3], [4], [6], [7], [8], [9], [10], [20], [21].

For a mixed CPU-GPU platform, Chi-Keung et al. proposed a new programming model called *Qilin*, and automatic load distribution system that considered the size of a data-set called *adaptive mapping* [3]. For automatic distribution, it first requires a training run to build a database of relationships between the size of a dataset and processing time. Users prepare CPU code, accelerator code, and special data arrays, which are described in

Qilin API. Then their system automatically balances distribution of computation between them while taking data size into account. Such features are similar to our "profile run" in Section 2.3. This did not include a system to determine the off-load parts as well as Ref. [4] or extract processing flow.

For a mixed CPU-FPGA platform, *DARES* [5] by Andrew Milakovich et al., *Hthread* [6] by Andrews et al., and *FUSE* [7] by Aws et al. are typical examples. DARES is one of a state-of-the-art software and hardware codesign framework on a reconfigurable system. Their target platform is a mixed FPGA-CPU platform called DARE, which is different from ours. In this framework, users first divide a target application into tasks and describe a communication between tasks in sequential manner. Then DARES compiles the tasks and the communication hardware. If suitable hardware modules for tasks are available, DARES uses them. This framework is similar to our "Backend," but DARES users have to re-compile a target application source code and profile it. The other two frameworks also hide arbitration and data communication, since users do not need to care about hardware modules on FPGA. They just write software source codes in a coventional manner, and then implemented parts are automatically replaced with pre-defined hardware modules. They do not focus on automatic choice of the parts or data transfer time either.

Most of them targets the expert user who can write code from scratch. However, as far as we know, there is no other similar work that can accelerate running binaries without accessing to the original source code. Most of common off-loading systems did not touch importance to function call graph with data and its data transfer time.

5.2 Related Techniques Used in Courier

In terms of the processing flow extraction, Feng et al. proposed a sophisticated method to extract fine grained dataflow from low-level program representation, and an algorithm to convert dataflow into threads-level parallelism [2]. They instrument a new static profiling path to GCC middle-end. Although their algorithm supports multi-thread, the target is a program code not a running binary.

DLL-injection or DLL hijacking are used in software research field [22], [23]. Purpose of them are fully replaces an original function. Moreover, data transfer and processing flow are not a matter. We use DLL hijacking as a basis for realizing dynamic off-loading, but we didn't directly use it. One of the important problems of today's application acceleration is data transfer time. Once we fully replaces an original function, data transfer time easily degrades performance. Furthermore, Courier has to maintain original processing flow after off-loading. Thus, we proposed Off-load Switcher to address these problems. In addition, DLL hijacking technique can be widely used in Linux. This means that Courier potentially support many platform. Heterogeneous platform which has CPU that runs Linux and accelerator emerged even in embedded area.

6. Conclusion

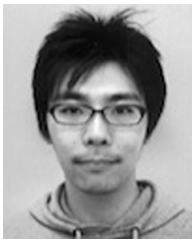
This paper presents Courier: a new toolchain for application acceleration. Courier is designed for detecting a processing flow of a target running binary and function off-loading without needing access to and re-compile the original source code of the binary. It is consisting of main three part: Frontend, Courier IR, and Backend. Frontend analyzes and detects processing flow within the running binary. Backend provides Function Off-loader which automatically replaces functions in the binary with corresponding accelerator functions, reduces the number of data transfer time, and maintains original processing flow. Courier IR generates a task graph and bridges Frontend and Backend. Finally, application binaries of HOG, dgemm and PSD are accelerated by using Courier on CPU-GPU environment.

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] GPU-Accelerated Libraries: available from ⟨https://developer.nvidia.com/-gpu-accelerated-libraries⟩.
[2] Li, F., Pop, A. and Cohen, A.: Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs, *Micro, IEEE*, Vol.32, No.4, pp.19–31 (2012).
[3] Luk, C.-K., Hong, S. and Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, *Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Vol.MICRO 42, pp.44–55 (2009).
[4] Becchi, M., Cadambi, S. and Chakradhar, S.: Enabling Legacy Applications on Heterogeneous Platforms, *The 2nd USENIX Workshop on Hot Topics in Parallelism*, pp.1–6 (2010).
[5] Milakovich, A., Gopinath, V.S., Lysecky, R. and Sprinkle, J.: Automated Software Generation and Hardware Coprocessor Synthesis for Data-Adaptable Reconfigurable Systems, *2012 IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pp.15–23 (2012).
[6] Andrews, D. and Peck, W. et al.: The Case for High Level Programming Models for Reconfigurable Computers, *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp.21–32 (2006).
[7] Aws, I. and Shannon, L.: FUSE: Front-end user framework for O/S abstraction of hardware accelerators, *International Symposium on Field-Programmable Custom Computing Machines*, pp.170–177 (2011).
[8] 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).
[9] Vahid, F., Stitt, G. and Lysecky, R.: Warp Processing: Dynamic Translation, *Computer*, Vol.41, No.7, pp.40–46 (2008).
[10] 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 (2004).
[11] Intel Developer Zone: Pin - A Dynamic Binary Instrumentation Tool, available from ⟨https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool ⟩.
[12] dlfcn.h - dynamic linking, available from ⟨pubs.opengroup.org/onlinepubs/007904-975/basedefs/dlfcn.h.html⟩.
[13] Dalal, N. and Triggs, B.: Histograms of Oriented Gradients for Human Detection, *International Conference on Computer Vision & Pattern Recognition*, Vol.1, pp.886–893 (2005).
[14] Automatically Tuned Linear Algebra Software (ATLAS), available from ⟨http://math-atlas.sourceforge.net/⟩.
[15] CUBLAS, available from ⟨https://developer.nvidia.com/cuBLAS⟩.
[16] GNU Octave, available from ⟨https://www.gnu.org/software/octave/⟩.
[17] Power Spectral Density Estimates Using FFT - MATLAB & Simulink - MathWorks, available from ⟨http://jp.mathworks.com/help/sig-nal/ug/psd-estimate-using-fft.html?lang=en⟩.
[18] FFTW Home Page, available from ⟨http://www.fftw.org/⟩.
[19] cuFFT, available from ⟨https://developer.nvidia.com/cuFFT⟩.
[20] 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, pp.386–391 (2011).
[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*, pp.1208–1213 (2008).
[22] Berdajs, J. and Bosnić, Z.: Extending Applications Using an Advanced Approach to DLL Injection and API Hooking, *Softw. Pract. Exper.*, Vol.40, No.7, pp.567–584 (online), DOI: 10.1002/spe.v40:7 (2010).
[23] Willems, C., Holz, T. and Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox, *IEEE Security and Privacy*, Vol.5, No.2, pp.32–39 (online), DOI: 10.1109/MSP.2007.45 (2007).



Takaaki Miyajima recieved his B.E. degrees 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 (M06) 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.

(Recommended by Associate Editor: *Hiroaki Yoshida*)