A Domain Specific Language and Toolchain for OpenCV Runtime Binary Acceleration using GPU

Takaaki Miyajima †, David Thomas †, Hideharu Amano †
† Graduate School of Science and Technology, Keio University, Japan
† Department of Electrical and Electronic Engineering, Imperial College London, United Kingdom
vision@am.ics.keio.ac.jp

Abstract—Computationally intensive applications, such as OpenCV, can be off-loaded to accelerators to reduce execution time. However, developing an accelerated system requires a significant amount of time, requiring the developer to first choose an accelerator and which parts to off-load, then to port and the offloaded kernels to the accelerator using many accelerator-specific tools. In addition to the low-level parallelism of the accelerator, the developer also needs to extract and utilize system-level parallelism found within the application, while making sure that the application still executes correctly. This paper presents Courier, a toolchain and a domain specific language for Runtime Binary Acceleration, designed to simplify many of the steps involved in accelerating an application. The Courier toolchain can extract dataflow from a running software binary file, explore the off-loaded execution time on an accelerator, and then actually accelerate the original binary. By utilizing Courier, both expert and non-expert users can easily extract system-level parallelism and decide which part should be off-loaded to accelerators in a mixed software-hardware environment, without special knowledge on the target application source code and accelerator architecture. In a case study an OpenCV application is accelerated by 2.06 times using Courier, without requiring the application source code or any re-compilation of the application.

I. INTRODUCTION

Although accelerators consisting of FPGAs and/or GPUs can enhance the speed of computationally intensive software libraries, their programming often requires time consuming efforts, even for a specialist. Most complicated programs such as those using OpenCV also require the co-operation of a host computer and accelerators, so the programmer also must take into account the overhead of data transfer between the host and accelerators. Extracting system-level parallelism from a running binary using an experimentally derived coarse grained dataflow and/or profile data is also a key problem, especially in a mixed software-hardware environment, as it allows the programmer to decide which parts of the target program should be off-loaded to the accelerator.

For computational intensive programs, there are various platforms consisting of only CPU, CPU w/ GPU, CPU w/ FPGA and so on. Most programs are developed and compiled on a CPU-only platform, then ported to accelerators. When the programmer tries to port it, they have to rewrite the source code and re-compile it for a target platform. The development of the accelerator code is often more complex and error-prone than creating general-purpose software, and also requires specialised knowledge of the underlying accelerator architecture. Improving productivity, decreasing period of trial-and-error, and easy program porting are important research topics in high performance computation with accelerators.

Runtime Binary Acceleration aims to provide universal acceleration that automatically off-loads the most time consuming part of an program’s computation at runtime. Similar to Just-In-Time (JIT) compilers, we use this term for all methods of off-loading computation at runtime without requiring the original source code. Note that Binary doesn’t means 1.0 file, but executable file with compile-time information. Although this idea has been partly introduced in existing acceleration research, most researchers have focused on how to connect or off-load to hardware modules, how to efficiently generate accelerator codes, and so on. On the other hand, little research attempts to integrate these topics so as to make it easy for a no-expert. This paper presents Courier and Trailblazer, a toolchain and DSL (Domain Specific Language) for Runtime Binary Acceleration.

This paper includes:

• Progress on the Courier, which automatically profiles and extracts dataflow from running software binaries, and converts it to the Trailblazer language description. It also captures a DFG and actually off-load designated function to GPU by using our Dynamic Off-loader. User can obtain support for select parts to be off-loaded and are not required manual tweaks or re-compilation of source codes or binaries.

• Trailblazer, a simple pure functional language intended to transparently accelerate. Easy describing and modifying the dataflow graph, allowing the use to view the DFG extracted by Courier, and then re-write it determine the off-load parts at runtime next time the binary is executed.

• Practical case study using practical OpenCV program and GPU, including extraction of dataflow and system-level parallelism from running binaries, and actually off-load and shorten processing time using Courier and Trailblazer.
II. RELATED WORK

There is a significant amount of existing research into runtime automatic off-loading systems, with most of them focussing on instruction-level optimization. However, there is less work on the extraction of system-level parallelism from existing applications, or the transparent acceleration of binaries without access to the original source code. For a FPGA, Stitt et al. proposed Warp Processing which takes advantages of reconfigurable logic for acceleration[1][2]. They count branch frequency within a running binary to determine hot codes. The Dynamic CAD module then compiles the code for the hot-spot, and automatically generates FPGA circuits, similar to a Just-In-Time compiler. For a heterogeneous platform, Becchi et al. [3] are proposed runtime automatic off-loader for heterogeneous nodes and focus on a data transfer scheduling. Their system is consist of two parts and similar to us, however, they didn’t mention how to determine off-load part.

III. COURIER

In a traditional design cycle of application acceleration such as Fig.1, hardware acceleration experts first spend a lot of time on profiling, in order to understand and extract system-level parallelism from the target application source code. They then implement it on accelerators, potentially requiring many iterations to ensure correctness and performance bottlenecks. Each cycle from the source code modification to measures processing time is not short, as just changing dataflow a little and measuring the new one is a time consuming task. Heterogeneous platforms incorporating CPUs, GPUs and FGAs is one of the most difficult platform to distribute

A. Model Usage

Fig.2 gives an example of using Courier for an accelerator implementation cycle. Courier is intended for use in a mixed software-hardware environment. Courier can be used as a substitute for many separate design tools in the traditional style, without moving backward and forward among many application programs. Courier integrates functions which can substitute traditional manual processes, for example, DFG generator, runtime dynamic profile, dataflow analyzer, accelerator module/function generator, and implementation of the actual off-loading at runtime.

Courier provides runtime binary acceleration allowing users to extract system-level parallelism, and selects the target parts implemented on the accelerator. Courier does not only analyses running binary, but also off-loads a designated part to the accelerator. Even non-expert users can easily compare and choose the proper part for off-loading to an accelerator, without detailed knowledge of the target architecture.

B. Overview

Fig.3 shows an overview of Courier and its four main parts: Frontend (Runtime Analyzer); Trailblazer; Courier Assembler; and Backend (Accelerator Implementer). The running binary on the middle left is a target application profiled in the Frontend. Then, the profile log is parsed by the Courier Assembler and converted to Trailblazer. User designate off-load part in Trailblazer. Finally, the Backend implements accelerator code and off-load designated part by using our Dynamic Offloader. It also measures processing/data communication time and generates a DFG based on input. In the following sections, we’ll go into more detail on each stage.

C. Frontend (Runtime Analyzer)

Frontend extracts source data, function name and destination data by using DTrace[4]. DTrace is function name based runtime dynamic program profiler, which can trace an arbitrary set of functions in an application. In Sect.VI, we configure it to extract OpenCV API functions. By using DTrace functions, Frontend can trace and extract ”raw” data from a running binary without needing access to the original source or for any sort of re-compilation. Thus, users can simply start their application as normal, and Frontend traces functions as they are called.

Dynamic program profiling is one of the key features of Runtime Binary Acceleration for extracting system-level parallelism, since static program profiling needs sophisticated pointer analysis, and often cannot obtain important properties such as processing time, actual data size, transfer time, frequency of branch taken, etc.
D. Courier Assembler

This part is the core of the Courier. It connects profiled data from Frontend, dataflow to the Backend, and modified dataflow from Trailblazer. All profiled data and properties are described in this assembler and translated into more user-friendly Trailblazer description. The Courier Assembler is a simple row-based language where data items (e.g., processing time, function name in software library, function type, function call depth, connectivity of dataflow, and device properties) are placed on a single line. This row-based style makes it easy to extend Courier itself.

E. Backend (Accelerator Implementer)

This part maps the modified DFG back into the running binary, measures processing time of off-loaded part including data transfer, and actually implements it on the accelerator. We propose Dynamic Off-loader to automatically off-load for OpenCV GPU API and explain it in Sect.V. Currently, core accelerator modules/functions are implemented beforehand, but all measurements are from actual processing time with real data. Although Backend does not yet support FPGAs.

IV. TRAILBLAZER

Trailblazer is a language for Courier, which allows users to easily modify dataflow and quickly simulate off-loading function. Its structure is a simple pure-functional language which uses only two basic types and concise statements. Functions in Trailblazer can take zero or more sources, and results are assigned into a single destination. This feature naturally fits most stream type applications such as OpenCV, since most of them have arguments of source image and destination image. At present, Trailblazer is not completed.

A. Off-load Functions

Trailblazer has some special functions for designating the off-loaded function. CPU to the accelerator, and acc2cpu is for the opposite direction. In the Sect.VI-A, VideoCapture dst data are off-loaded to the accelerator. The functions run and the data are stored in the accelerator as follows. acc2cpu is called again to bring back GaussianBlur dst data to CPU. In this case, cv::GaussianBlur is off-loaded, and runs on GPU. Trailblazer does not implicitly change device properties without users’ confirmation. volatileInput/Output are automatically called to

B. Types and Grammar

The current version of Trailblazer has two basic types, designed for use with OpenCV: courierMat and courierFunc. courierMat contains member variables of cv::Mat (OpenCV image data format), which don’t need to be explicitly declared in the source code. Similarly, courierFunc corresponds to OpenCV function, but it only contains function name and does not have to be declared.

The grammar of Trailblazer deliberately restricts the number of arguments that functions can accept. OpenCV functions are classified into three types: sink, unary and binary transform. Sink functions can only take one source, and produce no result. Unary transform always takes a single source and produces one result. Similarly, binary transform always takes two sources and one destination. On the other hand, courierMat may have multiple destinations, but can only be connected to a single source. Another restriction in this version is simple and easy: just lining up courierMat/Func in the order of dataflow. Then, Courier automatically detects causality and generates DFGs.

V. DYNAMIC OFF-LOADING TO GPU FOR OPENCV

OpenCV is a widely used open software library for computer vision applications. Most OpenCV functions perform computationally intensive image transformations, and have clearly defined input and output images, and so are ideal candidates for off-loading to accelerators.
When performing Runtime Binary Acceleration, for example to the GPU, Courier automatically generates a function wrapper, called Dynamic Off-loader, around the original OpenCV function, which is able to re-route the calls to the equivalent GPU functions. The behavior of this mechanism is as follows. At start-up Dynamic Off-loader stops the running binary, then hooks designated functions by using LD_PRELOAD, dlsym and dlopen, hooking functions for dynamic shared library loading. It then re-directs them to the wrapper, then continues the binary. Such mechanism can be applied to any binary without re-compilation, and is supported in most Linux environments. The equivalent GPU functions must be available beforehand, so we used the OpenCV GPU API functions.

A. Structure of dynamic off-loader for OpenCV

Dynamic Off-loader, a part of Backend of Courier, generates wrapper function which includes source/destination data transfer, re-route designated function and applies pre/post-processing to data that is required for GPU API functions. Fig.4 shows an example of the dynamic off-loader for the cv::Sobel function. In the figure, cv::gpu::cvtColor functions which changes image properties is inserted before and after the cv::gpu::Sobel function based on "Property Calculator". The overhead of transferring image depends on the image size, but changing properties is constant-time.

There were some big problems of automatic off-load in initial implementation. First one was the arbitrary off-loading decision, second one was to reduce data transfer time when series of functions are off-loaded, and third one is Function Type restriction. First and third problems was caused by LD_PRELOAD which hooks all the same name functions in target binary. In initial implementation, simply re-directing the software function to the GPU would mean all function-calls in the target would be off-loaded, rather than just the intended functions. Second problem became arise from that when series of functions are off-loaded, data transfer happens at each function and degrade performance. Function Type restriction keeps user to use off-load functions which have the same Function Type as original functions. It's still future work.

To off-load arbitrary functions, we introduced Off-load switcher to choose one of three possible states for a function: non-off-load, off-load and pass through. The switcher is on the top of Fig.4. Non-off-load keeps the target function runs on CPU, and Pass Through re-route the source image is just assigned to the destination images so as to invalid original function in binary. The condition of the switcher is judging from arguments of function or function ID which is contained in Courier Assembler. In the case study, there are two calls to cv::Sobel but we once tried to off-load only the y-axis function. Y-axis cv::Sobel are off-loaded to GPU by examining the function arguments at runtime.

To reduce data transfer time, Dynamic Off-loader hooks the head of a series of functions and runs all functions in that. For original functions in binary, the off-loader hooks and invalids them by Pass Through. In Sect.VI, successsive three functions(cv::GaussianBlur, cv::cvtColor and cv::warpAffine) can be off-loaded. Dynamic off-loader hooks cv::GaussianBlur and the following two functions also wrapped in that function. By using this technique, the number of data transfer has been reduced at once, compared to three of that of initial implementation.

B. Frontend for Analysing OpenCV Binary

We configured Frontend to extract the following data whenever the target application calls an OpenCV function: OpenCV API function/arguments name, function start/end time, function type, source/destination image data, and image size/bit depth/channels. Currently, this information is specific to OpenCV, but with simple modifications of the Frontend, many other library could be tracked in the same way. All the OpenCV functions have the prefix "cv::", such as cv::blur. Function type denotes one of three transform types (sink, unary, and binary), corresponding to the FunctionType variable in courierMat type. The source/destination image data are actual image contents. This means that data movement and modification outside OpenCV can be traced, even if the target running binary didn’t use OpenCV function to modify images such as "MissingLink".

VI. CASE STUDY

In this section, an image processing application example is used to illustrate our approach. This application is off-loaded and accelerated on mixed software-hardware environment by using Courier and Trailblazer. Experimental environment MacOS X 10.6.8 64bit, Intel Core i7 3.20GHz, The target GPU is GeForce GTX280 with PCI-Express Gen.2, and C++ OpenCV 2.4 binaries compiled with GCC ver.4.2.1 are used.

A. Target Running Binary

The target running binary is a pre-computation stage for video a stream, typical of those seen in computer vision area.
The binary uses three main features which are commonly seen in this kind of application: OpenCV API functions (Gaussian Blur, Sobel operator, Affine transformation), direct pointer access to the image, and image duplication via memcpy.

First, we explain the processing flow of this binary, shown in List.1, but note that neither Courier nor the user need access to the source code. The flow consists of three main steps:

1) A 1280x720 input image frame is retrieved from the video source, then smoothing (cv::GaussianBlur) and a gray scale transform (cv::cvtColor) are applied.

2) Two direction diagonal Sobel operators (cv::Sobel) are applied, by first rotating the image by 45-degrees (cv::warpAffine), then applying x/y-axis Sobel operators. The memcpy function (cv::copyTo) is also used to duplicate after the image shifting. Just after this point, the case study study application modifies an image by direct pointer access, as an example of an untraceable function by Courier. Such functions can be detected and handled by Trailblazer in co-operation with the user.

3) Then two images are combined(cv::add) into one image, converted to 8-bit scale (cv::convertScaleAbs), and then the histogram is adjusted (cv::equalizeHist). Finally the result image is displayed.

I. Described in Trailblazer:

After running Courier and wait certain period of time, the running binary is described in Trailblazer as shown in List.2. At line 7 the cv::VideoCapture function is used to provide input images (video stream) for the dataflow. In the actual software, cv::VideoCapture is given extra arguments which are captured in the Courier Assembler. However, when it is translated to Trailblazer, these are hidden from the user. Furthermore, this function was taken as argument of volatileInput special function at line 3, and protected from modification.

Listing 1: Psudo code of target running binary

```cpp
#include <opencv2/opencv.hpp>
int main(int argc, char **argv)
{
while(true)
{// Original Input
  VideoCapture cap >> input;
  Mat preProc = cv::cvtColor(input);

  // Step 1): Pre-processing
  Mat GB_dst = cv::GaussianBlur(input);
  Mat rotated = cv::warpAffine(preProc, 45deg);
  Mat addXY = cv::add(sobX_mod, sobY);
  Mat addXY8bit = cv::convertScaleAbs(addXY);

  // Step 2): Sobel Operator, two direction diagonal
  for (ptr = 0 to 100)
  { Mat sobX_mod.data[ptr] = sobX.data[ptr + 1]; }
  Mat duplicated = cv::copyTo(rotated);
  Mat sobY = cv::Sobel(duplicated, "Y-axis");

  // Step 3): Unite 2 Sobel image and adjust Histogram
  Mat eqHistDiag8bit = cv::equalizeHist(addXY8bit);
  cv::imshow(eqHistDiag8bit); }
}
```

Listing 2: Target running binary described in Trailblazer

```cpp
void courier_main(void)
{// Original Input/Output, unchangeable nodes
  volatileInput(Videocapture_dst);
  volatileOutput(equalHist_dst);

  // Step 1): Pre-processing
  VideoCapture_dst = cv::VideoCapture();
  GaussianBlur_dst = cv::GaussianBlur(VideoCapture_dst);
  cvtColor_dst = cv::cvtColor(GaussianBlur_dst);

  // Step 2): Sobel Operator, two direction diagonal
  warpAffine_dst = cv::warpAffine(cvtColor_dst);
  sobel_x_dst = cv::Sobel_x(warpAffine_dst, CV_16S, 1,0);
  sobel_y_dst = cv::Sobel_y(copyTo_dst, CV_16S, 0,1);

  // Step 3): Unite 2 Sobel image and adjust Histogram
  add_xy_dst = cv::add_xy(MissingLink_dst, sobel_y_dst);
  cvtScaleAbs_xy=cv::convertScaleAbs_xy(add_xy_dst);
  equalHist_dst = cv::equalizeHist(cvtScaleAbs_xy);
  cv::imshow(equalHist_dst); }
```

"MissingLink" is a user-defined inline function which doesn’t use the OpenCV API. Current versions of Frontend cannot trace such functions because they do not have a segment name in binary, but Courier can “guess” their existence based on execution time and source-function-destination causality when the binary is running on the CPU. Additionally, if such user-defined functions are not in-lined, Courier can trace their name and source/destination images.

II. Generated DFG:

In the automatically generated DFG shown in Fig.5, ellipse nodes and rectangle nodes represent images (cv::Mat) and functions, respectively. Processing time are displayed in the second row of ellipse node. Nodes are aligned in chronological order. Each images is processed in 41970 [us] in total, less than 23 frames per second([fps]).

The size of rectangle nodes reflects the proportion of node execution time to total execution time, allowing the programmer to easily recognize that large nodes (e.g. cv::GaussianBlur) occupy large fraction of total processing time. Courier traces the absolute time of each function start and end, so the processing time of OpenCV API functions includes any function setup time, as well as the actual pixel processing time. For example, cv::GaussianBlur includes both the Gaussian filter but also the time taken to create the Gaussian filter coefficients. Additionally, MissingLink has described as an dark node.

The size of ellipse nodes reflects the data size of im-
Fig. 5: Generated DFG running on CPU (with notations): Basic and simple system-level parallelism were extracted. MissingLink, user-defined function, was guessed by Courier.

image (height x width x bit-depth x channels). In this case study, there are four kinds of node sizes despite the fixed image size of 1280x720, as the cv::cvtColor, cv::Sobel and cv::convertScaleAbs functions change the number of channels or bit-depth of their source image, increasing or decreasing the memory size of each pixel.

Around the path from cv::warpAffine_dst to cv::add_xy, branching and converging show that both the x-axis and the y-axis Sobel operator are using the same image as a source, and these destination images became sources of cv::add, a binary transform. Furthermore, the vertical relative offset (separated by dash line) of two diagonal Sobel filters illustrates sequential execution on CPU, and is an opportunity to exploit simple system-level parallelism. Using this graphical representation, even non-expert users can easily understand the sequential part to be removed, and such analysis is only available due to runtime dynamic profiling on Frontend.

III. Accelerate with Courier and Trailblazer:
The current version of the Courier Toolchain supports GPU as an accelerator. In Courier for OpenCV, cpu2acc copies or moves designated images to the accelerator. The following images are stored and the functions run on the accelerator until acc2cpu function is called.

Step A. Search for Safely Off-loadable Part:
First, Courier searches for a safely off-loadable part, where source and destination images are both traced and an equivalent accelerated function is available – for example "Processing Step 1" in the case study. Courier suggests to the user that this part might be off-loaded. After the user’s confirmation, Courier automatically and dynamically generates wrapper and off-load this part (the details are in Sect.V), cv::GaussianBlur, cv::cvtColor and cv::warpAffine are replaced with pre-defined GPU version of OpenCV API, cpu2acc, cpu::GaussianBlur, cpu::cvtColor and cpu::warpAffine, respectively. In addition, Courier introduces the following lines in Trailblazer:

```
cpu2acc(cv::GaussianBlur, MOVE, gpu0); // Move to GPU
acc2cpu(cv::warpAffine, MOVE, gpu0); // Back to CPU
```
corresponds to target binary runs on CPU, "Courier" is the result of the case study, and "Manual GPU" is a manually implemented GPU version of the original. For the off-loaded part, Courier achieves almost the same result as the manual GPU version. User kept two cv::Sobel and cv::add runs on CPU, because data transfer time was greater than the decrease in processing time and current Function Type restriction (mentioned in Sect.V). Overhead of Dynamic Off-loader (Backend) is less than 150[us], judging from "Pass Through"ed functions.

VII. CONCLUSION AND FUTURE WORK

This paper presents Courier and Trailblazer for OpenCV. Courier is a toolchain for Runtime Binary Acceleration, including a runtime binary analysis, system-level parallelism extraction and actually off-loads codes on an accelerator. Courier can trace runtime binary without any source code tweaks or re-compilation. Furthermore, Our Dynamic Off-loader automatically generates wrapper function for accelerator. Not only analysis, but also easily modification of dataflow and actual off-load to GPU by using proposed Dynamic Off-loader are unique point. A domestic specific language, Trailblazer is a simple pure-functional language for describing dataflow of stream application and designating off-load part. Finally, practical case study has shown as well. In this practical case study, some functions are dynamically off-loaded and x2.06 speed-up has been achieved.

We have many research topics to realize real Runtime Binary Acceleration. Firstly, more flexible dataflow modifications must be handled by Trailblazer and Courier Backend. Introduce new image data which doesn’t exist in original binaries and use them arbitrary. FunctionType restriction must be removed as well. These require global image storage on Backend. Furthermore, parallelization pragma/system should be introduced for parallel execution.

REFERENCES


TABLE I: Processing time comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoCapture</td>
<td>2811</td>
<td>—–</td>
<td>—–</td>
</tr>
<tr>
<td>Processing Step1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2acc</td>
<td>—–</td>
<td>709</td>
<td>—–</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>18594</td>
<td>7738 (Total)</td>
<td>7288</td>
</tr>
<tr>
<td>cvColor</td>
<td>1997</td>
<td>1497 (Total)</td>
<td>146</td>
</tr>
<tr>
<td>warpAffine</td>
<td>6455</td>
<td>1211 (Total)</td>
<td>686</td>
</tr>
<tr>
<td>Processing Step2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>acc2cpu</td>
<td>—–</td>
<td>338</td>
<td>—–</td>
</tr>
<tr>
<td>Sobel (x-axis)</td>
<td>809</td>
<td>1176</td>
<td>—–</td>
</tr>
<tr>
<td>cpu2acc</td>
<td>—–</td>
<td>178</td>
<td>—–</td>
</tr>
<tr>
<td>MissingLink</td>
<td>2982</td>
<td>—–</td>
<td>—–</td>
</tr>
<tr>
<td>copyTo</td>
<td>—–</td>
<td>393</td>
<td>—–</td>
</tr>
<tr>
<td>Sobel (y-axis)</td>
<td>631</td>
<td>—–</td>
<td>1245</td>
</tr>
<tr>
<td>Processing Step3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add(x and y)</td>
<td>370</td>
<td>173</td>
<td>—–</td>
</tr>
<tr>
<td>convertScaleAbs</td>
<td>4961</td>
<td>—–</td>
<td>—–</td>
</tr>
<tr>
<td>cpu2acc</td>
<td>—–</td>
<td>130</td>
<td>—–</td>
</tr>
<tr>
<td>equalizeHist</td>
<td>2184</td>
<td>1965</td>
<td>—–</td>
</tr>
<tr>
<td>acc2cpu</td>
<td>—–</td>
<td>331</td>
<td>—–</td>
</tr>
<tr>
<td>Total</td>
<td>41970</td>
<td>21608</td>
<td>1748</td>
</tr>
<tr>
<td>Speed-up</td>
<td>—–</td>
<td>48.5%</td>
<td>57.7%</td>
</tr>
</tbody>
</table>