A Toolchain for Application Acceleration on Heterogeneous Platforms

Takaaki Miyajima

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

School of Science for Open and Environmental Systems
Graduate School of Science and Technology
Keio University

July 2015

Preface

Computationally intensive applications such as image processing or computational fluid dynamics can be sped up, once the critical parts are off-loaded to accelerators and pipelined on mixed CPU-GPUs/FPGAs platforms. FPGA and GPU are becoming larger and larger thanks to improvement of VLSI technology. They can deal with target algorithms more complex than before. However, the conventional work-flow of application acceleration is very time consuming even for experts who have specialized knowledge of the underlying accelerator architecture. This is because it contains many things to consider along with off-loading and most of these are done manually. The conventional work-flow often starts from understanding the original CPU-only program. At this stage, extracting runtime processing flow from a running binary is a key problem. Programmers experimentally extract the processing flow, so they often tweak the source code to analyze and re-compile it for a target platform. In the next step, they have to decide which parts should be off-loaded to the accelerator, while also taking into account execution time, parallelism, and data communication overhead of a target platform. Finally, they implement, optimize, and debug accelerator kernels in a trial and error manner. Some studies have been done to deal with this problem, for example High Level Synthesis for FPGA or higher programming language for GPGPU. Such highly abstracted programming languages automatically explore and utilize small level parallelism such as multiply or add. Their main target users are highly skilled programmers and they follow the above described work-flow. Easing and simplifying application acceleration while taking advantage of the features of a heterogeneous platform are important research topics. Therefore, this thesis proposes our new toolchains to lighten the burden of application acceleration on heterogeneous platforms.

We developed two toolchains for application acceleration, (*Courier* and *Courier-FPGA*) to deal with the problems. Both toolchains are intended for non-expert users who have neither expertise in nor special knowledge of target heterogeneous platforms. The goal of the toolchains is to provide an automatic application acceleration on popular heterogeneous platforms. We also proposed an inter-accelerator pipeline (IAP) on multiple mixed CPU-GPU platforms. IAP forms a task level pipeline among multiple GPUs.

Courier automatically extracts a processing flow from running software binaries, generates a task graph, and off-loads designated parts to GPUs on a mixed CPU-GPU platform. All the user has to do is just refer to a task graph and modify it if needed. Manual tweaks or re-compilations of target

source codes or binaries are not required. *Courier* consists of three main parts: *Frontend (Runtime Analyzer), Courier Intermediate Representation (IR), and Backend (Accelerator Manager). Frontend* analyzes running software binary, and detects dataflow, and then generates *Intermediate Representation (IR)* and a task graph. Finally, *Function Off-loader* in *Backend* replaces the designated parts with the pre-defined accelerator functions, and it off-loads them to an accelerator. *Courier-FPGA* is based on Courier, but the target platform is a single mixed CPU-FPGA platform. It builds a function-level pipeline structure between the hardware modules on an FPGA and software functions on a CPU automatically. IAP provides yet another implementation methodology for stream computation applications on a multiple CPU-GPU platform. Each task is assigned to each GPU among intra/inter-node works in a pipelined manner. Practical case studies are conducted to confirm the applicability of the proposed toolchains. Applications including image processing, matrix multiplication, and Fourier transfer are accelerated on three heterogeneous platforms.

Acknowledgements

First of all, I would like to thank my two supervisors, Professor Hideharu Amano and Lecturer David B. Thomas.

Professor Hideharu, Hunga-san, allowed me to research freely, and gave me a number of suggestions, opportunities, and so on. Thanks to him, I started my own research topic, studied abroad, published a book, and won some awards. Without his patient guidance, I could not finish my Ph.D. I am very proud to study under Hunga-san at Keio University.

Lecturer David gave me a brand-new research topic, a bunch of idea, and a new way of doing research. Weekly two hours debate was unforgettable time for me. Without his cool ideas, I could not even start my Ph.D. I was so lucky to study under David at Imperial College London.

I would like to express my special appreciation and thanks to my doctoral committee members, Professor Shingo Takada, Professor Hideo Saito, and Associate Professor Takahiro Yakoh. This thesis has not been possible without elaborate, extensive and integral comments by them. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

I would especially like to thank all of hlab members. All of you have been there to support me when I researched for my Ph.D thesis. I am also grateful to all the members of ASAP group. Dr. Yuri Nishikawa shows me a pleasure and difficulty of Ph.D. life. She helped me so much as a Ph.D. senior. Hirokazu Morishita always smiles at ASAP members, and taught me a fun of FPGA. Kenta Inakagata is fast thinker and fast worker. Sorry for running after your first date. Naoko Yamada's cynical jokes and 2ch talks make us lough to death. You two made my hlab life fun and meaningful. Toshiaki Kamata tells me a lot of things about a computer. From beneficial things to trivial ones. I saw a GPU night owl in Akihiro Shitara. You are a true GPU specialist. Takaaki Yokoyama carved out a unique path and it was cool. Takayuki Akamine helped me and discussed with me so much not only research but also heavy metal. You're a my best junior. I found out lack of experience while supervising Mao Hatto. Thank you for inviting me the home party! Naru Sugimoto and Takuji Mitsuishi impressed me with their passion for Zynq and GPGPU. I learned many thing from you. Masayuki Kimura - I decided to go to doctoral course since I was not in it with your passion for implementation. Yoshihiro Yasuda - He always entertain us. Thank you for bringing me my under wear. I also would like to thank my argumentative and cheerful members, Yoshiki Saito (Shacho-

san), Yu Kojima (a skiing Finance guy) and Yosuke Sakai (a respected Otaku), I was grad to pass the first year in entertaining conversation with you. Nobuaki Ozaki (Shiro-chan), Eiichi Sasaki (Sassa-), Kazuei Hironaka (Nyacom) Shimpei Nomura (a waver between GPU-BOX and NEC), and Seiichi Tade (a broad vision boy).

I want to thank my seniors for all their troubles, especially Dr. Yasunori Osana (University of the Ryukyus), Dr. Masato Yoshimi (The University of Electro-Communications), Dr. Toshihiro Hanawa (The University of Tokyo), It has been exciting to discuss with RECONF Kenkyu-kai members. They gave me constructive comments. Keisuke Dohi (Mitsubishi Electric) and Shinya Takamaeda (Nara Institute of Science and Technology) are tough competitors to me. But I'm grad to discuss, advice and drink with you.

My gratitude extends to Nachiket Kapre, Sumanta Chaudhuri, Gordon Inggs, Justin Wong, James Davis, David Borland, Josh Levine, Adam Powell, James Mardell, Kan Shi and Jirabhorn Pui at the Circuits and Systems Research in Imperial College London. I researched a lot, learned a lot, and talked a lot. It was a best time of my Ph.D. life. I also understand the key of strength of the United Kingdom. I want to go Imperial and research with you again.

Half a year internship at Imperial College London in 2011 is supported in part by a Grant-in-Aid for the Global Center of Excellence for High-Level Global Cooperation for Leading-Edge Platform on Access Spaces from the Ministry of Education, Culture, Sport, Science, and Technology in Japan.

The study of Task Pipelining in Section 3.3 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."

A special thanks to my parents, Keiichi and Atsuko. Words cannot express how grateful I am to mother and father for all of the great support. I'm sorry for worrying you, but I've got Ph.D :) At the end, I would like express appreciation to my beloved Azusa Sugasawara who encourage me. Please be next to me with your lovely smile.

Takaaki Miyajima
Chofu, Japan
July 2015

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Application acceleration on heterogeneous platforms which have a host CPU and accelerators such as GPU (Graphic Processing Unit) or FPGA (Field-Programmable Gate Array) have become important recently. This is because the increase rate of the processing power of typical single/multi-core CPU becomes slow.

This is because the processing power of typical single/multi-core CPUs has been increasing at a slower rate than before.

Heterogeneous platforms are used in a scientific computation domain which takes a lot of processing power and also used in an embedded device domain which requires a power efficient system. According to the TOP 500, a project that ranks and details the 500 most powerful computer systems in the world, five out of the top ten super computers use accelerators in 2014 [1]. Furthermore, the performance share of accelerators became more than 30% of sum of TOP 500 machines. Figure 1.1 shows the changes of performance share of accelerators in TOP 500 [2]. In the domain of embedded device, computing platforms which have host CPU with FPGA or GPU has become available [3] [4]. The difference between both domains is whether accelerators are discrete or not. In the domain of scientific computation, they exist independently and are connected via PCI-Express bus. On the other hand, they are integrated into one chip and are connected via proprietary bus in the domain of embedded device. Especially in the domain of scientific computation, multiple node which have GPU becomes common. Each heterogeneous node is connected via optical fiber cables and uses Message Passing Interface (MPI).

Another reason why heterogeneous platforms has become more important is improvement of the programming environment. Useful libraries such as BLAS for GPU or OpenCV for FPGA are available [5] [6]. New programming languages for such platforms are also provided. OpenCL [7] is a good example of new cross platform language. It can make a program for CPU, GPU and FPGA. Not only OpenCL, there are many researches of programming language for the heterogeneous

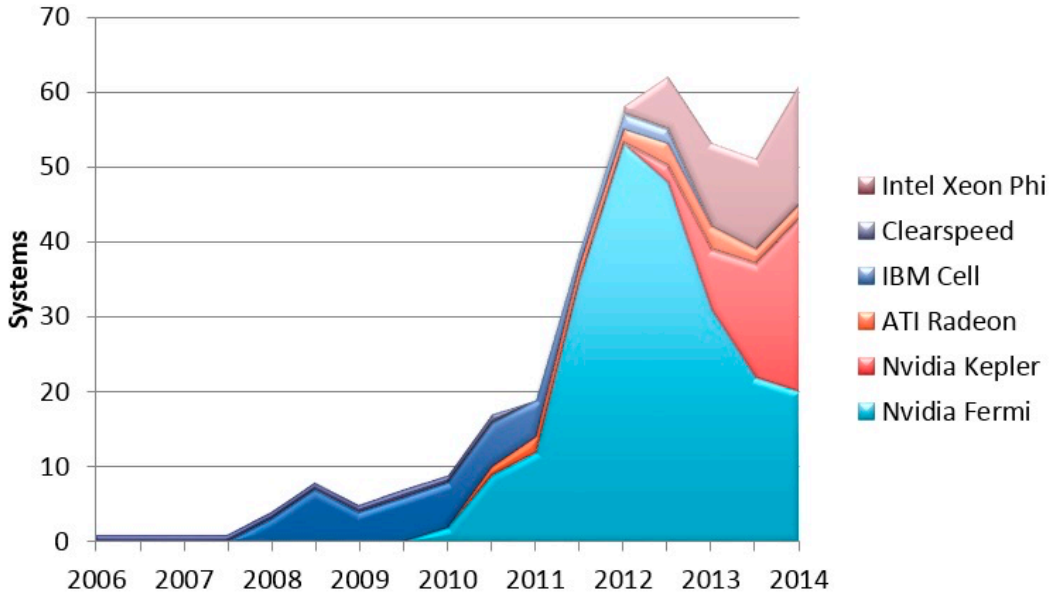


Figure 1.1: The changes of performance share of accelerators in TOP 500 [2].

platforms [8] [9] [10]. Users of these languages need to re-write the code causing the bottle neck part in their target applications. Hence, target users of such researches should have special knowledge of the target platforms, programming languages, and applications.

Although the combination of heterogeneous platforms and new programming environments has become mainstream, it is not easy to accelerate existing applications without expertise or special knowledge. Users of legacy or undocumented applications do not have enough knowledge of the executing processing flow, and often they do not have the source code itself. For such users, performance improvement with accelerators have been almost impossible. If the users understand the flow of the applications 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. Reflecting such a situation, there is a need for a new toolchain which can treat the binary of existing application and off-load critical parts onto accelerators without user intervention.

1.2 Objective

The objective of this thesis is to propose a new toolchain that provides simple work-flow of application acceleration for non-expert users. Three heterogeneous platforms are chosen, a node which has a CPU and a GPU (single-node mixed CPU-GPU platform), multiple nodes which have a CPU and GPUs (multi-node mixed CPU-GPU platform), and a node which has a CPU and an FPGA (single-node mixed CPU-FPGA platform).

As briefly mentioned in the previous section, most researches on developing work-flow of off-loading focus on programmers who understands the source code and want to improve its perfor-

mance. For developing the code of the accelerator which performs the same function of the target code with much higher performance, various types of tools and languages have been proposed. They help the analysis of the source code [11], accelerator management [12] [13], and accelerator kernel implementation [14] [8] [15] [16]. Unlike them, I do not intend to generate the accelerator code itself. It is assumed that the target application programs use a common library like OpenCV, BLAS or FFT, and the corresponding library codes of the accelerator are already available. Our target users do not need to have the source code of acceleration target. Our toolchain extracts the call flow of the functions, and find the parts which can be off-loaded to the accelerator during execution of the binary. The current version of the toolchain cannot do anything if the target binary does not include corresponding accelerator functions. Even with this limitation, the proposed toolchain can help many legacy code users who are not a target of the conventional work-flow.

1.3 Contribution

The main contribution of this thesis is to propose a new toolchain for application acceleration called *Courier* and *Courier-FPGA*. Both automatically analyses specific functions and data in a running binary and replaces functions with corresponding accelerator functions if possible. Whole work-flow doesn't require user intervention. Target heterogeneous environments are a single-node mixed CPU-GPU platform, a multi-node mixed CPU-GPU platform, and a single-node mixed CPU-FPGA platform.

Courier is a new application acceleration toolchain for single/multiple-node mixed CPU-GPU platforms. It does not require original source code, manual tweaks, or re-compilation of the target binary, without user intervention. The users just have to refer to the result and modify off-load parts if needed. To realize these features, two main methods are proposed.

One is an automatic processing flow graph generation method of analyzable functions from a running binary.

This method includes tracing sub-programs to analyze functions and a heuristic approach to detect causality. The other is 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-loaded 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. In the case of multi-node mixed CPU-GPU platform, Courier makes an intra-node pipeline by using a parallelization library and a commonly used communication library. Each stage of the pipeline is composed of analyzed functions. Additionally, Courier uses a direct accelerator-to-accelerator communication system called PEACH2 in order to shorten the communication latency between each accelerator.

Courier-FPGA is another version of Courier that targets is a mixed CPU-FPGA platform. Courier-FPGA treats software functions on a host CPU and multiple hardware acceleration modules imple-

mented 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. *Pipeline Generator* builds the pipeline in which processing flow is the same as the original one even if the original flow is not pipelined.

Practical case studies are shown so as to demonstrate the applicability of Courier and Courier-FPGA. For Courier, three practical applications are used: 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, 8.96 and 1.23 times by using the existing GPU functions on a mixed single-node CPU-GPU platform. For Courier-FPGA, another three practical applications are used: HOG, Motion Tracking, and 3D modeling. HOG was sped up 3.98 times on the existing hardware modules by making a mixed software hardware pipeline on Xilinx's Zynq platform. Two other cases were also sped up 22.1 times and 1.29 times, respectively.

1.4 Thesis Organization

The organization of this thesis is illustrated in Figure 1.2. It is described that three heterogeneous platforms and their programming environments in Chapter 2. A mixed single CPU-GPU platform, a mixed multiple CPU-GPU platform, and a mixed CPU-FPGA platform are shown. Then Courier is presented, a toolchain for application acceleration on a mixed CPU-GPU platform in Chapter 3. Courier is composed of three main components; Frontend, Courier IR, and Backend. Courier has platform specific features in Backend. Function Off-loader is for a mixed single CPU-GPU platform, and Task Pipeline is for a mixed multiple CPU-GPU platform. Case studies are given to show the capability of Courier. In Chapter 4, Courier-FPGA is presented, a toolchain for application acceleration on a mixed CPU-FPGA platform. Courier-FPGA shares Frontend and Courier IS with Courier, but Backend is different so as to make best use of the CPU-FPGA platform. Courier-FPGA's Backend has a Pipeline Generator that generates a mixed software hardware pipeline. Case studies are also shown in order to show the applicability of Courier-FPGA. Finally, I conclude the thesis.

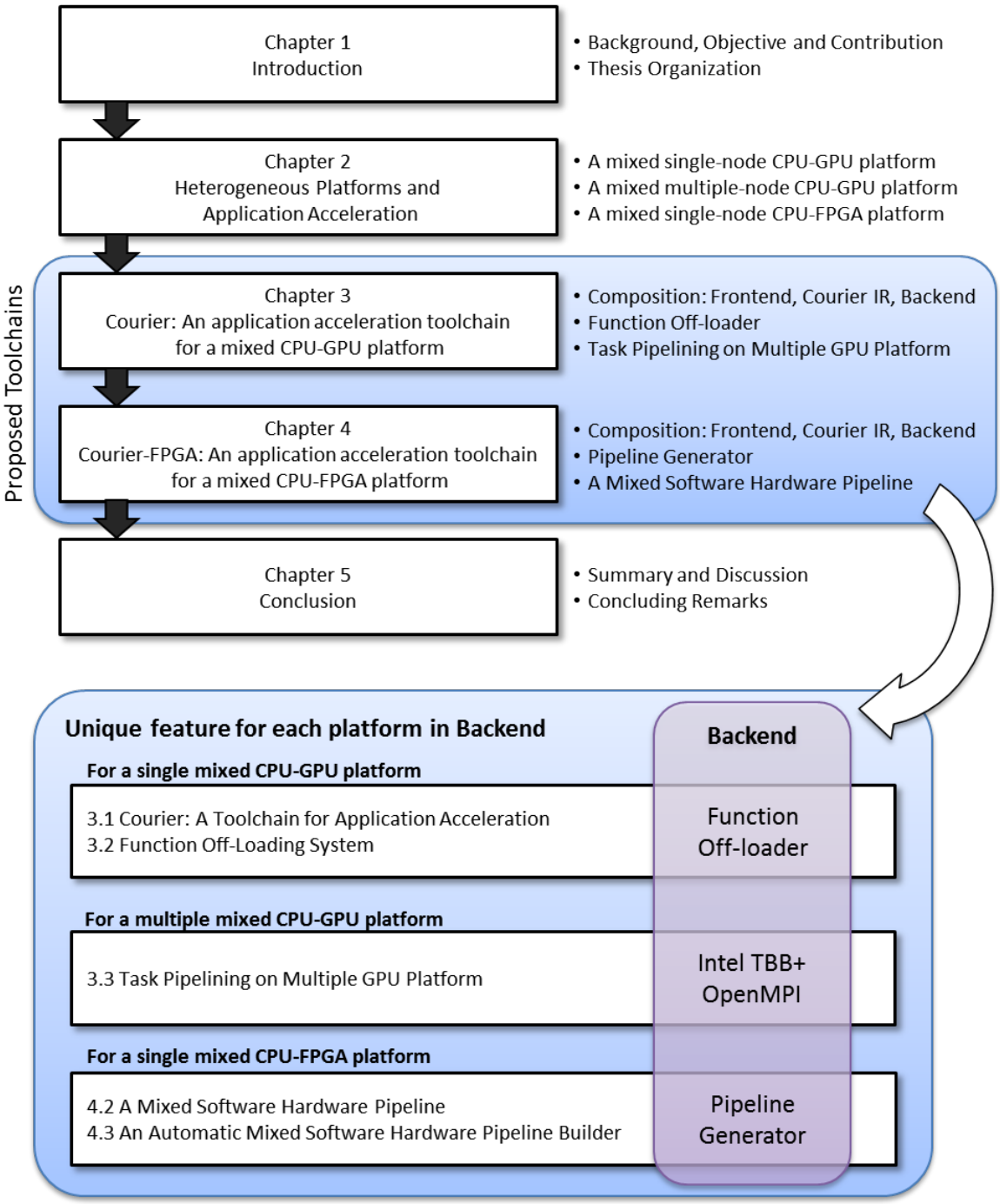


Figure 1.2: Thesis organization.

Chapter 2

Heterogeneous Platforms and Application Acceleration

Heterogeneous platforms have a host CPU and one or more accelerators in a node. This chapter overviews three main heterogeneous platforms, including characteristics of accelerators, and application acceleration work-flow. An acceleration by using the accelerators on such platforms is called "off-load". First, single-node mixed CPU-GPU platform is describe in Section 2.1. In Section 2.2 and 2.3 present a multi-node mixed CPU-GPU platform, and a single-node mixed CPU-FPGA platform. Then, it is explained that the conventional application acceleration work-flow on these heterogeneous platforms.

Figure 2.1: Example of a single-node mixed CPU-GPU platform. A home-built computer.

Figure 2.2: Example of a multi-node mixed CPU-GPU platform. TCA by Univ of Tsukuba [17].

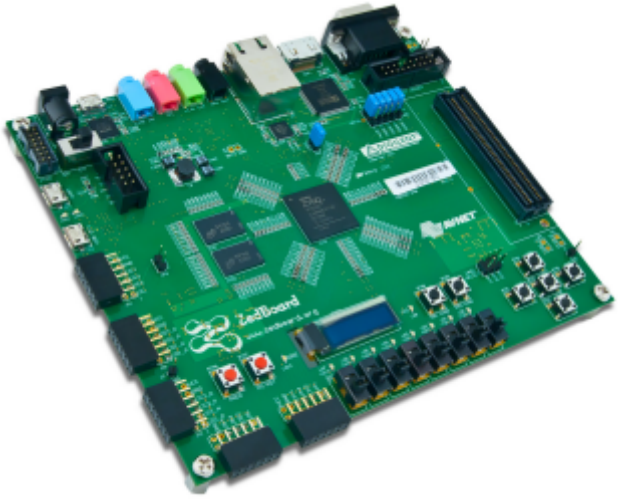


Figure 2.3: Example of a single-node mixed CPU-FPGA platform. Zedboard by AVNET.

2.1 Single-node mixed CPU-GPU platform

This subsection explain a single-node mixed CPU-GPU platform by taking NVIDIA's Graphic Processing Unit (GPU) for example. Software and hardware aspects of GPU is described. Then the structure of the CPU-GPU platform is explained.

2.1.1 GPU: Graphic Processing Unit

Figure 2.4: NVIDIA's Graphic Processing Unit. (GeForce GTX Titan [18])

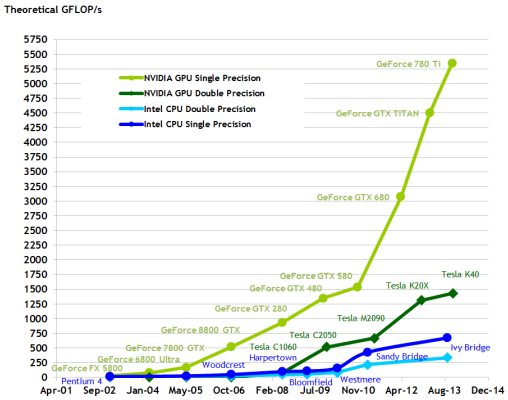


Figure 2.5: Comparison of peak performances of Intel CPU and NVIDIA GPU [19]

Graphics Processing Unit (GPU), shown in Figure 2.4, was originally designed for image processing or computer graphics. It has thousands of processing cores since image processing has very high data parallelism. GPU are currently used in many other areas including scientific computation. This kind of usage is called "General Purpose computation using GPUs (GPGPU)". GPGPU gains advantages over CPU in terms of peak performance, cost-performance ratio, and power-performance ratio. GPGPU also promotes a multi-node mixed CPU-GPU platform.

Figure 2.5 compares between FLoating Operations Per Second (FLOPS) of Intel's CPU and NVIDIA's GPU since 2003 [19]. According to this figure, the performance of GPU has advanced faster than that of CPU. This is because each core of GPU has come to contain an SIMD unit or super function unit.

2.1.1.1 Hardware aspect of GPU

GPU is different from CPU in many aspects. As this thesis previously mentioned, GPU is designed for massive data parallel processing. Consequently, most transistors are used for data processing, unlike the CPU, which has spare transistors for instruction control. Figure 2.6 shows an architecture of a current GPU (Kepler generation). The small green boxes are processing cores (called CUDA Core), and small orange boxes next to green boxes are L2 caches. The architecture of the GPU has a hierarchical structure. CUDA Core is a basic processing unit, and it has a floating point unit, integer processing unit, logic unit, move/compare unit, branch unit, and so on. A Streaming Multiprocessor

Figure 2.6: An architecture of GPU. Small green boxes are processing cores. [19]

(SM or SMX) is a group of CUDA Cores, and the GPU is a group of SMs. The number of CUDA Cores in SM is different for each GPU generation. In the case of the previous generation, Fermi, it was 32. In the case of the current generation, Kepler, it is 192. Additionally, the number of SM is different for each product, and the processing power of each product is almost equal to the number of SMs.

NVIDIA's GPU architecture employs a Single Instruction stream Multiple Thread (SIMT) model. In the SIMT model, multiple independent threads are executed concurrently using a single instruction. This is closely-linked to GPU's programming model.

Memory Hierarchy Figure 2.7 shows a memory hierarchy in a GPU. There are three layers: a per-thread local memory, a per-block shared memory, and a global memory. Each CUDA thread can access multiple layers during a program execution. Local memory is used for registering each CUDA thread. Local memory and shared memory can be accessed quickly since they are on-chip memories. On the other hand, global memory is slow since it is an off-chip memory. In terms of size, local memory and shared memory are small, while the global memory is large. One of the key points of GPU programming is reducing the access to global memory and increasing the access to local or shared memories. Additionally, a global memory contains two read-only areas, called "constant memory" and "texture memory", which are used for storing constant values in a general processing and texture values in a computer graphics, respectively. [20]

Memory of GPU programming can be summarized as follows. Memory spaces of the host (CPU) and device (GPU) are separated in the CUDA programming. Moreover, GPU has a hierarchical memory. Programmers need to keep this in mind and take advantage of this structure to accelerate their programs.

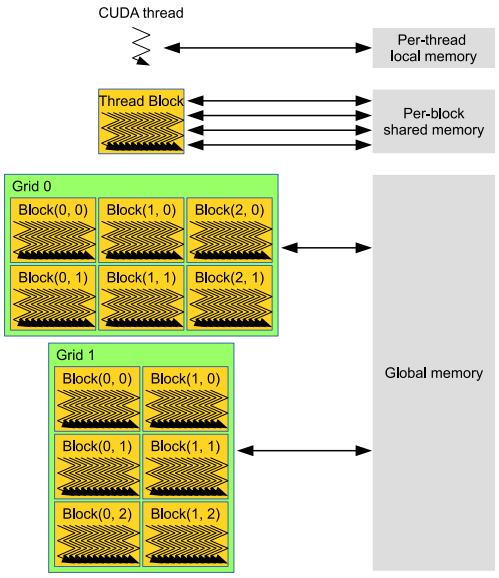


Figure 2.7: Memory hierarchy in a GPU. Local memory is the smallest but the fastest. Global memory is the largest but slowest. Shared memory is in between [19].

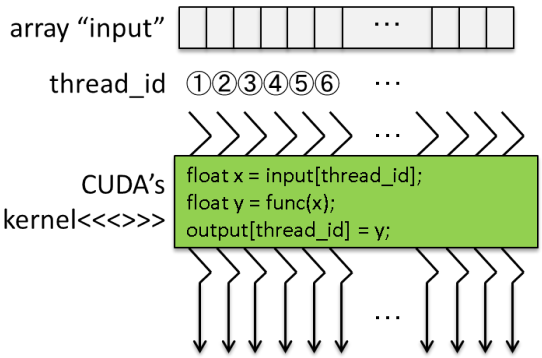


Figure 2.8: Concept of parallel threads in the CUDA's programming model.

2.1.1.2 Software aspects of GPU

CUDA: Compute Unified Device Architecture NVIDIA's GPU has its own software environment called Compute Unified Device Architecture (CUDA). CUDA is a scalable parallel programming mode and software platform. It is not a graphic API but a C/C++ based programming language. [20] A program running on a GPU is called a "CUDA Kernel" or simply a "Kernel". Additionally, a program running on a CPU is called "host program". Each kernel is kicked by a host program, and input and output data are stored in a memory on a GPU. Memory spaces of CPU and of GPU exist separately. Thus, data in the GPU memory space cannot be accessed by the CPU and vice versa. A typical program first sends the input data from CPU memory space to GPU memory space.

CUDA Thread Figure 2.8 shows the concept of parallel threads in the CUDA's programming model. Each thread has a unique ID (threadID) that executes the same CUDA kernel, but the input data are different. Input data depend on the ID, and the branch condition depends on input data. Thus, it is classified as SIMT. Correspondence between physical structure and software model is as follows. CUDA Core corresponds to CUDA Thread, SM corresponds to Thread Block, and GPU itself corresponds to Grid. CUDA introduces a hierarchical thread management mechanism. A certain number (up to 1024) of CUDA threads are packed into a "thread block". CUDA threads in the same thread block are executed on the same SM. This is illustrated in Figure 2.9. CUDA threads in a thread block are divided evenly into 32 threads by a hardware scheduler in GPU. Each of the 32 threads is called a "warp" and assigned to SM. SM can execute up to 16 warps. A certain number of thread blocks is

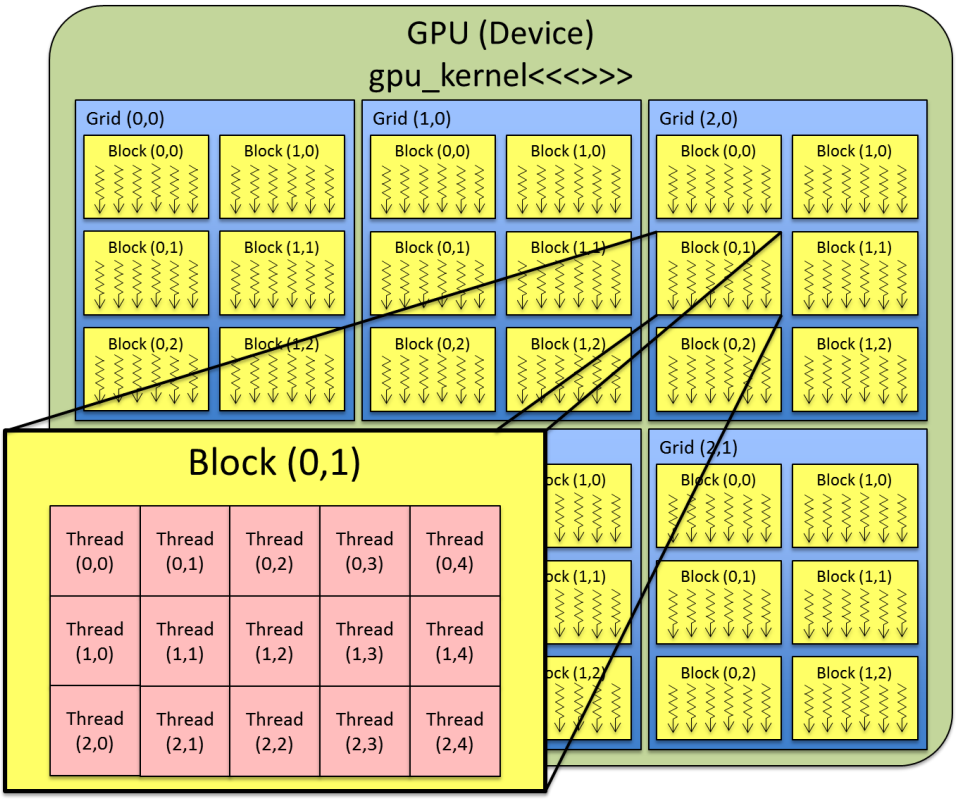


Figure 2.9: CUDA threads are packed into a *Thread Block*, and thread blocks are packed into a *Grid*.

packed into a "grid". The grid is assigned into a GPU when a CUDA kernel is launched. Additionally, the grid can contain thread blocks out numbering CUDA cores in a GPU. A hardware scheduler in a GPU assigns and executes threads blocks in sequence. This is similar to a batch process.

Multi GPU programming CUDA provides some special functions for multi GPU programming on a single node. Unified Virtual Address (UVA) unites separated memory spaces. UVA hides memory management between a CPU and a GPU from the users in a single node platform. Thanks to this software mechanism, the users do not need to explicitly write the codes that copy the data. A GPU can send the data to another GPU directly without CPU intervention. Memory copy functions of inter-node in a source code can be replaced by just pointer interacts. Data copied from/to a CPU and GPUs can be hidden, but data copies are restricted by an interconnection. The slowness of CPU-GPU communication is still one of the big problems of this platform.

2.1.2 Platform structure

An ordinary mixed CPU-GPU platform has a host CPU and a single GPU. The host CPU and GPU are connected by PCI-Express, which is a standardized serial bus for the computer. In the case in Figure 2.10, a Generation 3's 16-lane bus is used and its maximum bandwidth is 15.75 GB/s. The overview of processing flow is as follows. The users first prepare the initial input data on the CPU,

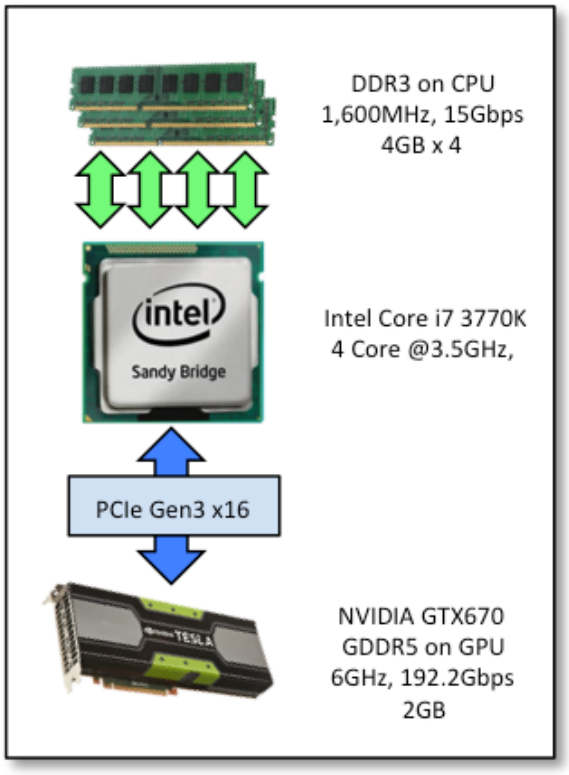


Figure 2.10: Example of CPU-GPU platform used in Section 3.4.

and send them to the GPU's global memory, and then GPU processes the data. Finally, GPU sends back the result data to the CPU.

2.2 Multi-node mixed CPU-GPU platform

This subsection explain a multi-node mixed CPU-GPU platform by taking University of Tsukuba's HA-PACS Tightly Coupled Accelerators (TCA) for example. It is described that the structure of this platform since all nodes are the same as those for a mixed CPU-GPU platform. Then, a HA-PACS TCA including a special direct GPU-GPU communication system is explained.

2.2.1 Structure of an ordinary mixed CPU-GPU cluster

A multi-node mixed CPU-GPU platform becomes common in the area of high performance computation. Each node that has CPUs and GPUs is connected via optical fibre cables. One of the most significant problems with this platform is data transfer among nodes, as illustrated in Figure 2.11. Each node is connected by optical fibre cables, and data transfer between nodes is controlled by the host CPU. This means that the data must be going through the CPU's memory when the GPU wants to communicate with the GPU on another node. Especially in the small data transfer, this problem is serious since such data require lower latency rather than wider bandwidth.

This platform is well known to enormously speed up data parallel applications. Such data parallel applications are implemented on multiple GPUs as follows. A user first exploits data level parallelism (DLP) and distributes data to each GPU, and then the same computation is applied to the data in all the GPUs. Finally, result data are sent back to the host CPU. If other computations are left, the result data are distributed and processed again. However, the bottleneck of this type of implementation is bandwidth to distribute the data. Since each GPU requires different data, the required memory bandwidth linearly increases along with the number of accelerators. When the users want to increase the number of GPUs or nodes, communication latency and data bandwidth between GPUs over the nodes become critical. Current CUDA API partly supports direct communication between GPUs within nodes not between them. To communicate with other GPUs in different nodes, multiple memory copies via the CPU memory are required. These redundant memory copies cause an increase of latency that severely degrades performance, especially in the case of small data.

Task level pipelining is a way to implement streaming applications on a multiple mixed CPU-GPU platform. It does not increase required bandwidth along with the number of GPUs, since the same amount of data transfer is required between tasks each of which is implemented in a GPU. A task level pipeline utilizes the data level parallelism (DLP) inside a task in a single GPU and runs tasks with multiple GPUs in a pipelined manner. A user first exploits task parallelism and assigns each task to a GPU. GPUs are basically connected in a line to realize the corresponding task flow. Then the data are processed along with the task flow on GPUs. A task level pipeline on multiple accelerators is common in an embedded application that uses FPGA, but has not been tried with GPU. The problem is that the redundant data copy is required to send data to the another GPU in another node. Especially, small data transfer is often needed.

To deal with the problem of communication among GPUs over multiple nodes, Tightly Cou-

pled Accelerators (TCA) and PEACH2 board have been proposed [21]. TCA is capable of low latency communication between GPUs over different nodes. Unlike other non-direct communication, it can eliminate redundant memory copy overhead. PEACH2 is a board provided in ordinary supercomputer nodes to realize TCA. The key feature of low latency is that PEACH2 enables direct communication via the standard PCIe protocol, thus protocol conversion overhead is eliminated.

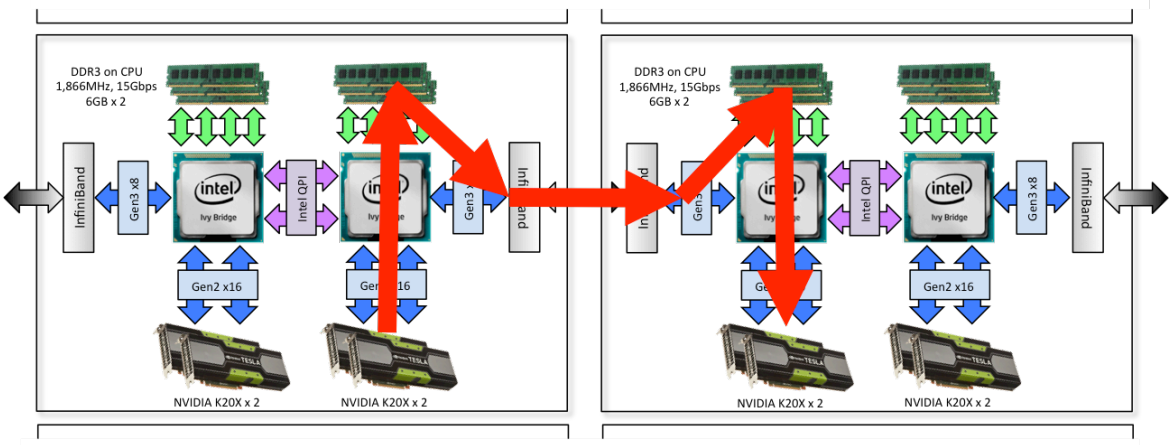


Figure 2.11: Communication via ordinary optical fibre cables. Data need to be sent to CPU memory once, then go to CPU memory on the other node, and finally go to the GPU.

2.2.2 HA-PACS TCA

The Highly Accelerated Parallel Advanced system for Computational Sciences (HA-PACS) is the latest generation of the PACS/PAX series supercomputer at the University of Tsukuba. HA-PACS is the demonstration system for parallel computing with Tightly Coupled Accelerators (TCAs). TCA realizes direct communication among multiple accelerators over computation nodes thanks to PEACH2. In this environment, each computation node (TCA node) has two Ivy Bridge-EP processors, four GPUs, and one PEACH2 board. Specification of used TCA node are shown in Table 2.1. By using PEACH2, each GPU is connected via PCIe Gen2 x8 with small input/output overhead. Figure 2.12 depicts an overview of a TCA node. By taking advantage of PEACH2, on the left of the figure, up to 32 GPUs can directly communicate with another GPU in another TCA node. This bunch of TCA nodes is called Node Cluster. Up to 16 TCA nodes can constitute a node cluster, and multiple Node Clusters are connected via InfiniBand.

A heterogeneous supercomputer cluster that has CPU, GPU, and FPGA has been researched. AXEL [22] from Imperial College London has one with multiple CPUs, GPUs, and FPGAs. These three computational units are conjoined via PCI. From the viewpoint of a CPU, FPGAs and GPUs are equivalent to an accelerator. In other words, the FPGA is just used for an accelerator, not a communication system. This thesis discuss the difference in application implementation in Section 2.4. QP [23] from Illinois University is more similar to TCA node. It has two CPUs, four GPUs, and one FPGA. However, the FPGA is just used for an accelerator as well as AXEL. In this respect, a

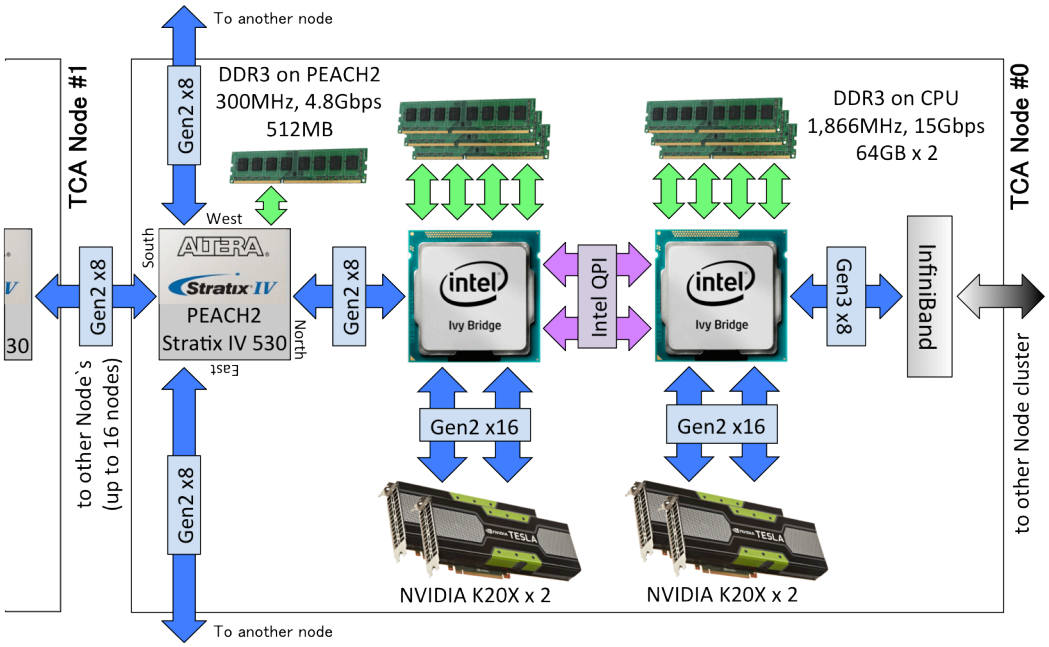


Figure 2.12: Physical overview of TCA Node: Each TCA node has two CPUs, four GPUs, one PEACH2 board, and one InfiniBand.

Table 2.1: Specification of used TCA Node

OS	Scientific Linux 6.3 (kernel 2.6.32)
CPU	Intel Xeon E5-2680v2 @ 2.8GHz (IvyBridge-EP) × 2
# of Cores	20 Core/Node (10 Core/Socket × 2 Socket)
Frequency	2.8 GHz
Peak Performance	224 × 2 GFLOPS/Node
PCI-express	Gen.3 × 80 Lane (40 Lane/CPU)
Memory	128 GB, DDR3 1866MHz,
GPU	NVIDIA Tesla K20x
# of GPUs / Node	4
Single Precision	14.36 TFLOPS/Node, 3.59 TFLOPS/GPU
Double Precision	5.24 TFLOPS/Node, 1.31 TFLOPS/GPU
CUDA Core	10752/Node (2688/GPU)
SMX	56 (768 CUDA Core)/Node, 14 (192 CUDA Core)/GPU
Memory	24 GByte/Node (6 GByte/GPU)
PEACH2	Altera Stratix IV GX 530 FPGA
InfiniBand	QDR×2 rail
Others	PEACH2, InfiniBand (QDR×2)
Connection	(Mellanox ConnectX-3 dual head)

TCA node with PEACH2 that can be used as communication system and accelerator is different from previous research. Additionally, AXEL and QP must return the output data to CPU memory once, even if the user just wants to transfer the data to another accelerator. The TCA realizes ultra low

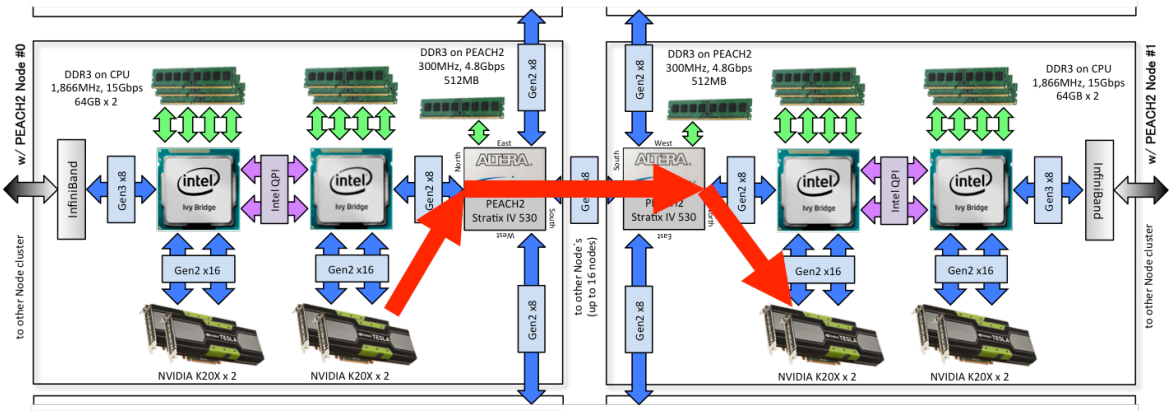


Figure 2.13: Communication via PEACH2. Data are sent via PEACH2 and directly goes to the GPU. PEACH2 enables ultra low latency direct communication among multiple accelerators within Node Cluster.

latency direct communication between accelerators. That is, the TCA is more suitable for stream computation than AXEL or QP.

Although a TCA can take many topologies, mesh/torus, hyper-cube, or ring, this thesis explain a special ring topology used in current TCA clusters. All eight nodes are connected by a ring network, and two ring networks are directory connected.

2.2.2.1 PEACH2: PCI-Express Adaptive Communication Hub 2

PEACH2 is a key technology of HA-PACS TCA. Communication latency must be reduced to fully utilize multiple accelerators, since the I/O bandwidth bottleneck seriously degrades performance. PCI-Express Adaptive Communication Hub 2 (PEACH2) is an FPGA switch that provides an ultra low latency direct communication via PCI-Express [21]. In the current version, up to 32 GPUs within 16 nodes can communicate less than 2.5μsec at minimum thanks to PEACH2. Researchers call such accelerators Tightly Coupled Accelerators (TCA). This section describes the architecture of PEACH2 and the TCA used in our proposal.

As shown in Figure 2.14, the PEACH2 board uses Altera's Stratix IV GX530NF45 FPGA as a switch core. It has four PCIe Gen2 x8 lanes, a DMA controller, a Nios II soft core processor, and a DDR3-SDRAM memory. Figure 2.15 depicts the block diagram of PEACH2. Four PCIe ports and a DMA controller are implemented as hard IPs and connected via an Avalon Streaming (Avalon ST) interface. The DMA controller, the core of PEACH2, can deal with multiple descriptors in arbitrary positions in order to realize continuous transfer. In addition, DDR3-SDRAM can be accessed in one clock cycle. FPGA logic utilization is less than 25%, and running frequency is 250MHz. Stratix IV GX has 32 Gigabit Transceiver Block (GBX) transceivers, all of which are used for the four PCIe lanes. Typically, the north port is connected to accelerators in the host node, and east/west/south ports are connected to another PEACH2 in other nodes. PEACH2 can be used to build various network topologies. Available topologies are fully connected mesh topology with four TCA nodes, cube

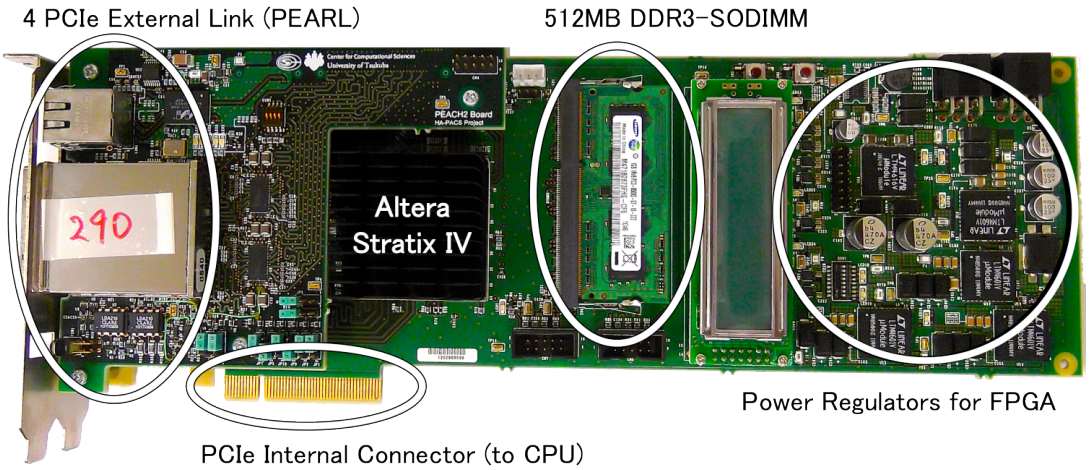


Figure 2.14: PEACH2 Prototype Board. It has Stratix IV FPGA, four PCIe Gen2 x8 lanes, a DMA controller, a Nios II soft core processor, and a DDR3-SDRAM memory.

Table 2.2: Specifications of PEACH2 board

Device Family	Stratix IV EP4SGX530	Usage [%]
Frequency	250MHz	—
Logic utilization	23%	23%
Combinational ALUTs	59k/424k	14%
Total block memory bits	2,763k/21,233k	13%
DSP block 18-bit elements	4/1,024	1%
Total GXB RX/TX PCS	32/32	100%
DDR3-SDRAM	512MB, 300MHz	—

topology with eight TCA nodes, or ring topology with 16 TCA nodes.

A direct communication network system has been researched. APEnet+ [24] is one of the closest in idea to PEACH2. Unlike PEACH2, which is compatible with PCIe protocol, APEnet+ uses its own protocol to realize direct connection between GPUs and network interface. PEACH2 has an advantage over APENet+ since it is not required to convert protocols. The CUDA 5 programming environment for NVIDIA GPU provides the RDMA mechanism to the GPU memory, called GPUDirect Support for RDMA, with a Kepler-class GPU [20]. This mechanism enables zero-copy communication between the InfiniBand Host Adapter and GPUs [25]. Although PEACH2 also uses the same RDMA mechanism, PEACH2 can eliminate the overhead for protocol conversion from PCIe to the InfiniBand packet, and the overhead of an MPI protocol stack.

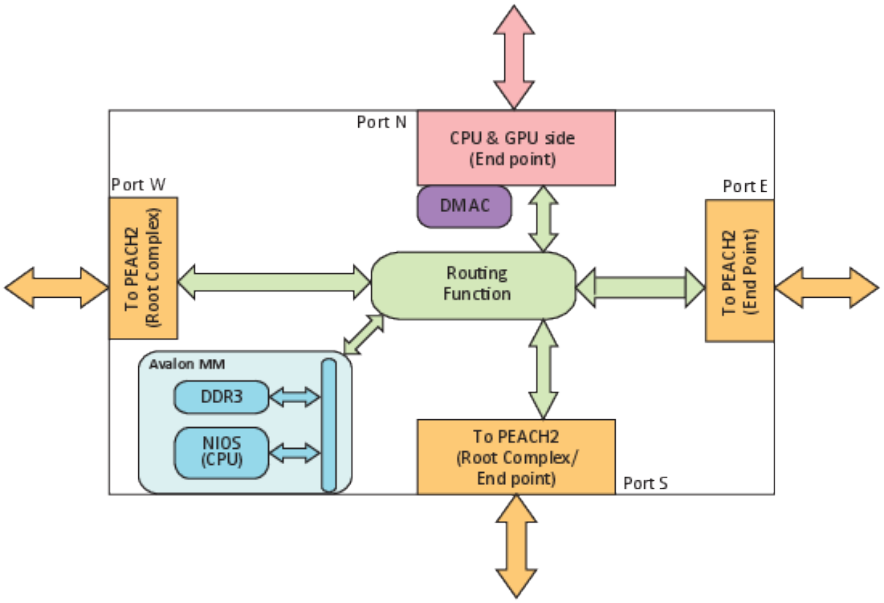


Figure 2.15: Block diagram of PEACH2. Four PCIe interfaces are implemented on GBX. DMA controller is implemented with hard-wired logic. It also has an Avalon interface to connect Nios II and DDR3 memory.

2.3 Single-node mixed CPU-FPGA platform

This subsection explains a mixed CPU-FPGA platform by taking Xilinx's Zynq-7000 All Programmable SoC [3] as an example. It is described that hardware aspects of FPGA and then explain the structure of a CPU-FPGA platform.

2.3.1 FPGA: Field-Programmable Gate Array

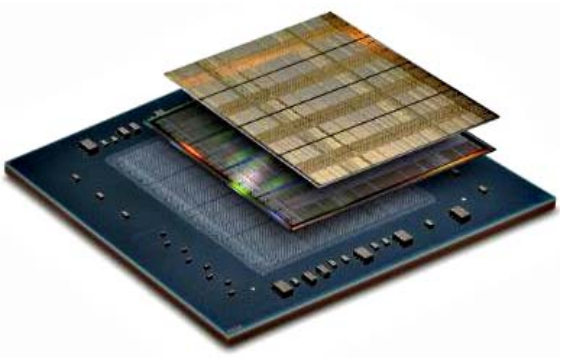


Figure 2.16: Xilinx's Virtex 7 Series FPGA. (XC7V2000T)

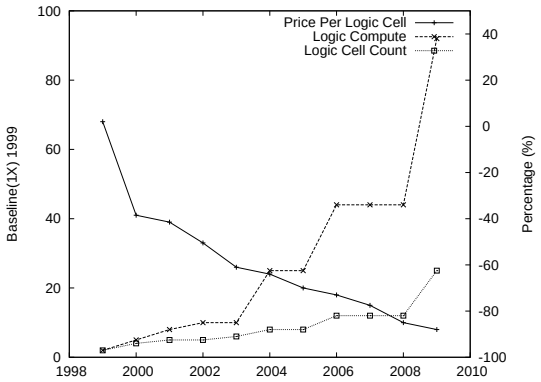


Figure 2.17: Historical advancement of FPGA components.

Field-Programmable Gate Array (FPGA) is a programmable integrated circuit designed to make a user-defined circuit. As the process technology advances, vendors manufacture denser chips containing multi-million of transistors [26]. Consequently, FPGA's range of applicability becomes wider. Figure 2.17 shows the advance in density, advance in speed, and reduction in price over time. One of the most important feature is that the FPGA does not adopt a von Neumann architecture. It does not need a program counter, an instruction memory, or software. On the other hand, the FPGA is less flexible than a CPU since it basically has hard-wired logic.

2.3.1.1 Hardware aspects of FPGA

FPGAs consist of three fundamental components: Logic Block, Connection Block, and Switch Block. Logic Block includes a Look Up Table (LUT), Configurable Logic Blocks (CLBs), Digital Signal Processing (DSPs) blocks, Block RAM, and Input/Output Blocks (IOBs). CLBs and DSPs are similar to a processor's arithmetic logic unit (ALU) but programmable. They can be programmed to perform arithmetic and logic operations like add, multiply, subtract, and compare. Unlike processors, which have fixed ALU designed in a general-purpose manner to execute various operations, the CLBs can be programmed with the operations needed by an application. This results in increased computing efficiency. Figure 2.18 shows a typical (island-style) FPGA architecture.

Logic Block can perform bit-wise, integer, and floating point operations. The results of the operations are stored in the registers present in CLBs, DSPs, or Block RAM. These blocks within an

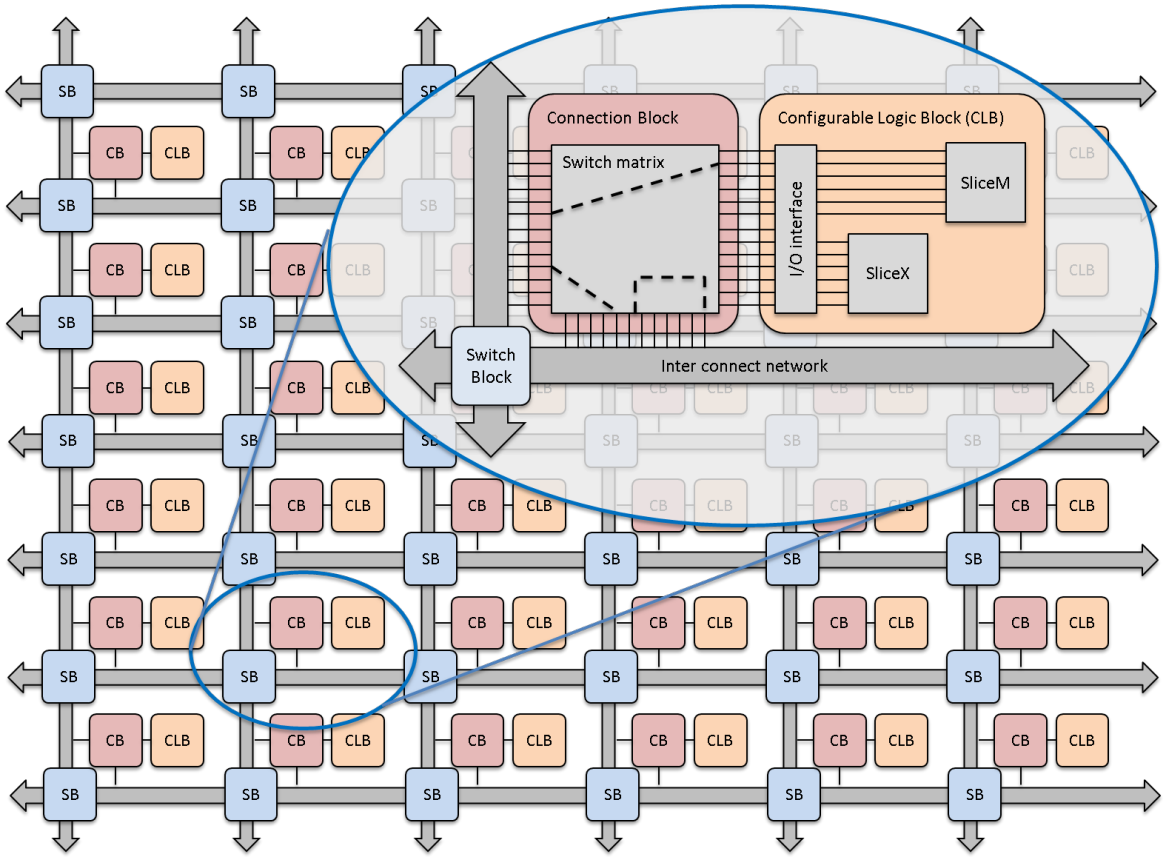


Figure 2.18: Typical architecture of an FPGA. Logic Blocks, Connection Blocks, and Switch Blocks are connected.

FPGA can be connected via flexible configurable interconnects. The output of an operator can be directly forwarded to the input of the next operator. That is, the FPGA architecture can be used to design data flow processing. A circuit is implemented in an FPGA by programming each logic block so as to play the role of a small logic portion. Each I/O block (not shown in the Figure 2.18) is programmed for either an input pad or an output pad. The routing is configurable to make all the necessary connections among logic blocks and from logic blocks to I/O block. The FPGA architecture provides the flexibility to create a massive array of application specific ALUs that enable both instruction and data level parallelism. Because data flow between operators are implemented with CLBs and DSPs, there are no inefficiencies such as processor cache misses. These operators can be configured so as to be connected with point-to-point dedicated interconnects, thereby a pipeline can be formed between multiple operators.

Look-Up Table Function generators in Virtex 7 FPGAs are implemented as six-input Look-Up Tables (LUTs). There are six independent inputs and two independent outputs for four function generators in a slice. The LUTs can implement an arbitrarily defined six-input Boolean function. Each LUT can also implement two arbitrarily defined five-input Boolean functions, as long as they share

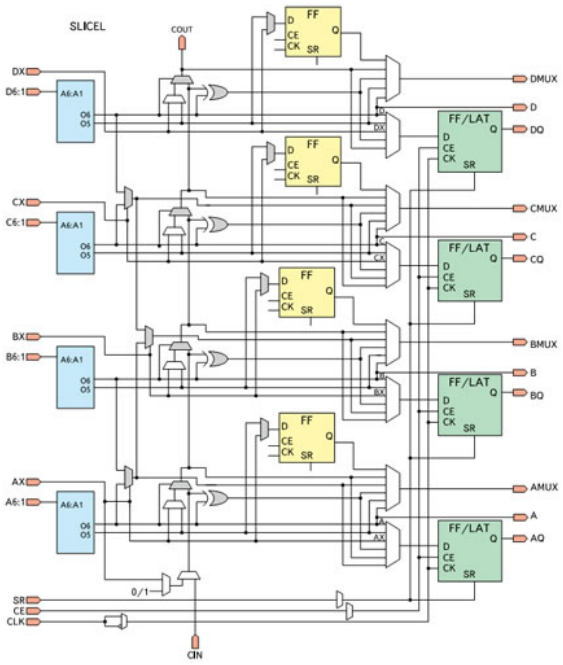


Figure 2.19: Architectural overview of SLICEL [27].

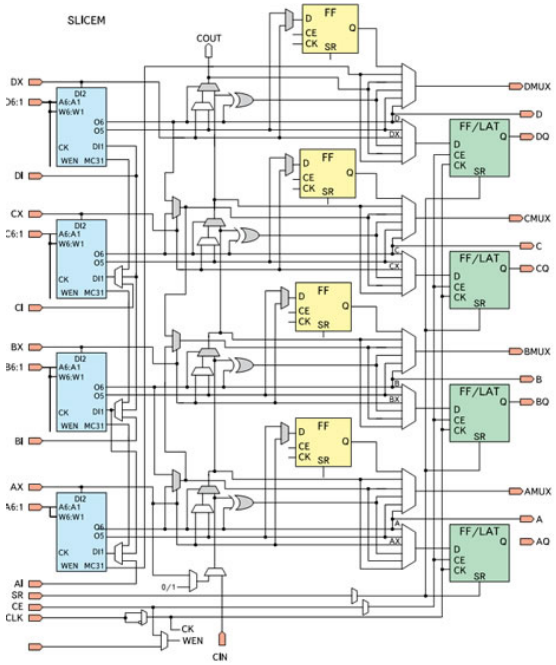


Figure 2.20: Architectural overview of SLICEM [27].

common inputs. In practice, a Hardware Description Language (HDL) is used to describe the digital circuit, and then synthesis tools are used to map the textual description into LUTs. The designer never defines logic in LUTs directly. The important consideration for a designer is representing a circuit efficiently to utilize available resources.

Slice Every slice contains four logic-function generators (or look-up tables); eight storage elements (D flip-flop), wide-function multiplexers, and carry logic. In addition to Boolean logic, a slice can be used for arithmetic and storing data. Some slices are connected in such a way that they can be used for data storage as distributed RAMs. This is accomplished by combining multiple LUTs in the slice. Slices can be used as shift registers. A shift register can delay an input with certain clock cycles. By using a single LUT, data can be delayed up to 32 clock cycles. By cascading all four LUTs in one slice, the delay can increase to 128 clock cycles. This enables small buffers to be instead of large block RAMs. There are two type of slice, SLICEM and SLICEL. SLICEL,, which is shown in Figure 2.19 can be combinational logic. “L” means logic. SLICEM, which is shown in Figure 2.20, can be combinational logic, distributed memory or shift registers. “M” means memory [27].

Configurable Logic Blocks The Configurable Logic Blocks (CLBs) are the main logic resources for implementing both sequential and combination circuits. Each CLB element is connected to a switch matrix for accessing the general routing matrix as shown in Figure 2.18. A CLB element consists of a pair of slices. These two slices do not have direct connections to each other, and each slice is organized as a column. Each slice in a column has an independent carry chain. For each CLB, slices

in the bottom and top of the CLB are labeled as SLICE(0) and SLICE(1) respectively.

Block RAM Block RAM is a dedicated random access memory grouped together in 36 Kbits blocks in Virtex 7 FPGAs. Every Virtex 7 FPGA has 156 to 1064 dual-port block RAMs. Each block RAM has two completely independent ports that share nothing but the stored data. The clock controls each memory access, read and write. All inputs, data, address, clock enables, and write enables are registered. Nothing happens without a clock. The input address is always clocked, retaining data until the next operation. An optional output data pipeline register allows higher clock rates at the cost of an extra cycle of latency. During a write operation, the data output can reflect the previously stored data, the newly written data, or remain unchanged. One common use of block RAMs in FPGA design is for FIFOs or data queues.

DSP Slices DSP applications use many binary multipliers and accumulators implemented in dedicated DSP slices. All Virtex 7 FPGAs have many dedicated, full custom, and low power DSP slices. In Virtex 7, the DSP slices are known as the DSP48E1 (48 bit DSP element) slices. Each DSP48E1 slice consists largely of a dedicated 25×18-bit two's complement multiplier and a 48-bit accumulator. They can operate at 600 MHz in maximum. The multiplier can be dynamically bypassed, and two 48 bit inputs can be connected into a single-instructions-multiple-data (SIMD) arithmetic unit, or a logic unit that can generate any one of 10 different logic functions of the two operands. The DSP48E1 slices provide extensive pipelining and extension capabilities that enhance speed and efficiency of many applications, even beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O register files. The accumulator can also be used as a synchronous up/down counter.

2.3.1.2 Software aspects of FPGA

Hardware Description Language (HDL) is used for implementing a desired circuit on an FPGA. The programmers write, verify a HDL and synthesize it by using an electronic design automation (EDA) tool. The EDA tool performs a place-and-route, timing analysis, simulation, design rule check, and so on. After the EDA tool finishes the whole process, a bit file is generated. The bit file is transferred to an FPGA via a JTAG cable or placed on an external memory device like an SD Card. Finally, FPGA is re-configured by the bit file.

Verilog and VHDL are the most popular HDLs for FPGA. They can treat a low level circuit and describe timing-specific behavior. On the other hand, the design complexity becomes a key problem as an FPGA becomes larger. The programmer must write detailed parts of an entire circuit and spend a long time verifying and debugging. It is often pointed out that this resembles the development by assembly languages. Furthermore, this complexity hinders exploration of design space such as pipeline depth. The programmers are required to prepare test benches to verify their design. A typical verification process includes simulation and debugging processes. Simulating the design takes a lot of

time since it is a clock-by-clock simulation. Debugging the design requires special knowledge since the output of simulation is wave-form. The design process from describing the design to debugging requires repetition of trial-and-error.

There have been many attempts to reduce the design complexity. High-Level Synthesis Language (HLSL) and IP cores are the most popular approaches. Vivado HLS and OpenCL are good examples of HLSL which is provided by the two biggest FPGA vendors: Xilinx and Altera. They are based on C/C++ language and an abstract hardware layer of FPGA. The compiler generates HDL from the HLSL. Thanks to this automatic code generation, the programmer does not need to pay attention to the clock or details of FPGA architecture. IP core is a sort of pre-defined library of software languages. FFT or floating point arithmetic operators are easy to acquire and are widely distributed by third-party IP suppliers. OpenCores [28] is one of the most successful projects to provide open-source IP cores. The programmers can obtain an AES cryptography processing module, Ethernet controller, and so on. The problem of IP core is timing adjustment. The users must manually meet the timing between a user logic and IP core. Besides, the combination of HLSL and IP core is the most powerful solution for reducing the design complexity. In the case of image processing, either an optimized HDL design exists [29], or a hardware module corresponding to each function can be easily generated by using recent high level synthesis tools for FPGA [6] [30].

2.3.2 Structure of a mixed CPU-FPGA node

In 2012, Xilinx released a new FPGA device called Zynq [3]. Zynq combines a dual-core ARM Cortex-A9 processor and Xilinx's latest FPGA logic fabric (Virtex 7). ARM and FPGA are implemented on the same die, realizing a flexible, low-power, and programmable system-on-chip (SoA). Zynq is comprised of two main parts: a Processing System (PS) formed around a dual-core ARM Cortex-A9 processor (upper-left area in Figure 2.21), and Programmable Logic (PL), which is equivalent to that of an FPGA (yellow area in Figure 2.21). It also has integrated memory, peripherals, and high-speed communications interfaces. ARM CPU and Virtex 7 FPGA are connected via a set of nine Advanced eXtensible Interface (AXI) buses. An AXI bus can be used for module-to-module communication on an FPGA part. By combining CPU and FPGA, Zynq takes advantage of many good points.

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 implemented in reconfigurable logic. To meet the performance requirements of recent advanced applications, legacy code working in embedded CPUs must be accelerated by the FPGA part.

2.3.2.1 ARM Coretex-A9 processor

Zynq's dual-core ARM Cortex-A9 processor includes a set of additional processing units called Application Processing Units (APUs). APU is composed of peripheral interfaces, cache memory,

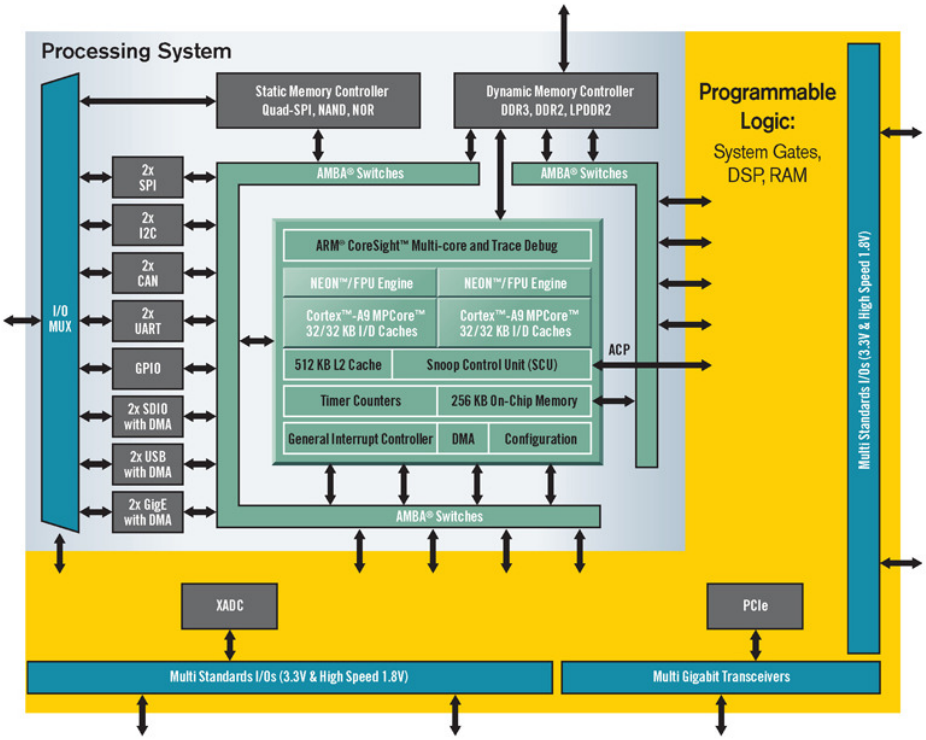


Figure 2.21: Architecture of Zynq-7000. ARM CPU and Xilinx FPGA are tightly coupled by AMBA bus [3].

memory interfaces, interconnect, and clock generation circuitry [31].

The maximum operating frequency of ARM Cortex-A9 is 1GHz, and each core has its own Level 1 data cache and instruction cache. Both caches are 32KB and used for storing frequently required data and instructions in order to achieve fast access and optimal performance. The two cores have a 512KB shared Level 2 caches. There is a Snoop Control Unit (SCU) that maintains coherency of L1/L2 cache the two cores. It also bridges the two cores and the Level 2 cache. SCU monitors accessed address lines and checks whether the lines are cached or not. Then it issues a write invalidate protocol when a write operation is observed in a location that a cache has a copy of, and the cache controller invalidates its own copy of the snooped memory location. There is a Memory Management Unit (MMU), and its primary role is to translate between virtual and physical addresses.

ARM Coretex-A9 also has a NEON engine that provides Single Instruction Multiple Data (SIMD) operations [31]. The NEON engine is used for image processing, a software defined radio that operates on a large number of data samples (pixels) simultaneously, and inherently parallel, generic signal processing functions such as Finite Impulse Response (FIR) filters and Fast Fourier Transforms (FFTs). NEON instructions are prepared as an extension to the standard ARM instruction set. The programmers can uses them explicitly or implicitly. The ARM compiler use these instructions when applicable parts are found in the source code. SIMD operations as NEON engine enhance the performance by accepting multiple sets of input vectors . Each element of input vectors is applied with the same operation simultaneously. The NEON engine supports many data types including

signed and unsigned integers, single precision floating point, and half-precision floating point. However, double precision is not supported. In addition to NEON, there is a Floating Point Unit (FPU). This unit is a sort of hardware accelerator and performs floating point operations in compliance with the IEEE 754 standard. It also supports single and double precision formats, with some additional support for half-precision and integer conversion.

2.3.2.2 Interconnection

ARM CPU and Virtex 7 FPGA are connected via an AXI bus. AXI is a member of the ARM AMBA family of micro-controller buses. The AMBA protocol is an open standard on-chip interconnect specification, allowing the connection and management of many controllers and peripherals in a multi-master design. It was originally developed by ARM for use in micro-controllers. Xilinx optimized and integrated the protocol for use within FPGA architectures. The main features of AXI are as follows; address/control phases are separate from data phases; byte strobes enable unaligned data transfers; burst-based transactions are possible with only the start address issued; read and write data channels are separate allowing low-cost Direct Memory Access (DMA); and transaction can be completed out-of-order. AXI version 4 is the latest standard and intended for the high-performance interface, suited for memory mapped I/O. It also supports burst transfer up to 256 data cycles per address phase. AXI4 introduces other two interfaces for specific situations.

- AXI4-Lite: This is a light-weight variant of the interface, used for memory mapped single transactions. The benefit of this protocol is a smaller logic footprint with a simplified interface. This variation does not support burst data and so only provides a single data transfer per transaction.

- AXI4-Stream: This does not have address bit. Thus, this is not memory mapped and allows for an unlimited data burst size. A single channel is defined for the transmission of streaming data, modeled after the Write Data Channel but allowing bursts of an unlimited amount of data. Connection is from master to slave only, so if bidirectional transfers are required both peripherals must be type of master/slave.

AXI's master and slave have a single data channel that can send/receive the data from/to slave or master. On the other hand, the situation of the address channel is different so as to support burst-based transactions. Read transaction uses a single channel and write transaction uses two channels. Write transactions have an additional write response channel when all data arrives from master to slave. Write response channel is also used for the slave to signal completion of a write transaction. Figure 2.23 and Figure 2.22 show the difference between a write transaction and a read transaction. AXI adopts a flexible Master-Slave protocol as well. For example, multiple AXI masters can have multiple AXI slaves. Thanks to this feature, multiple modules can be connected to multiple modules on an FPGA.

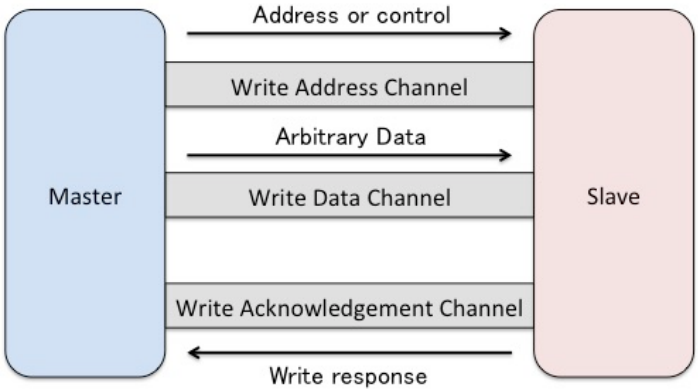


Figure 2.22: Three channels are required for write transaction

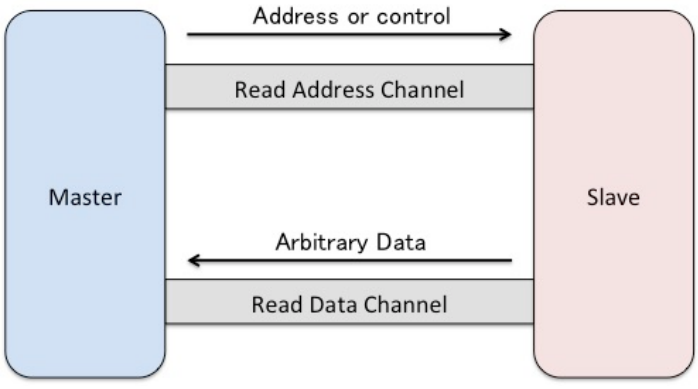


Figure 2.23: Two channels are required for read transaction

2.4 Application acceleration and programming environments

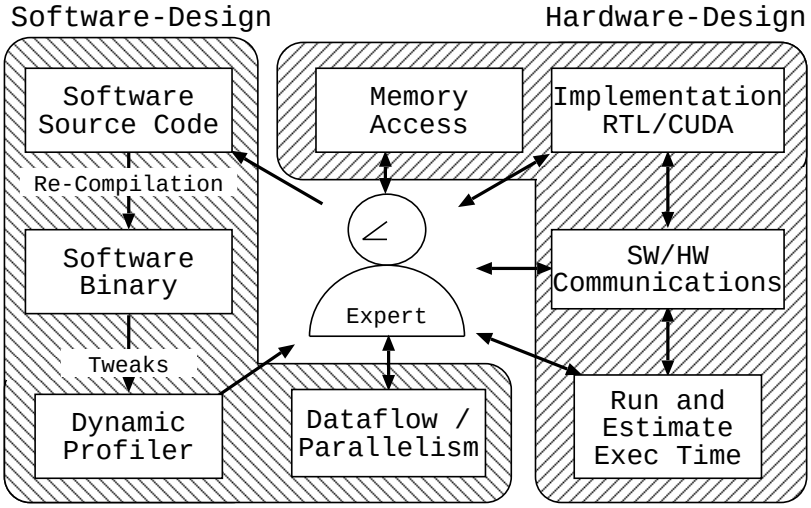


Figure 2.24: Conventional Acceleration Work-flow. Experts must make much effort, programmers must pay attention to many separated things, and the process must be done manually.

For expert programmers, performance of computationally intensive applications can be relatively easy to improve by off-loading time consuming parts to accelerators like GPUs or FPGAs. In contrast, users of legacy or undocumented applications do not have enough knowledge about how to execute the processing flow and often do not have the source code itself. For such users, performance has been almost impossible to improve with accelerators.

A conventional work-flow of application acceleration is shown in Figure 2.24. Rectangular boxes contains important elements when the programmers implement an accelerator code, and arrows denote manual processes. In the software design flow on the left, hardware acceleration experts first spend a lot of time on static/dynamic program analysis to understand and extract system-level parallelism from the target application source codes or binaries. They then decide which parts should be off-loaded and start implementation. The hardware design flow on the right potentially requires many iterations to ensure correctness and overcome performance bottlenecks. For data transfer between software and hardware , it is not easy to determine which part should be used, since data transfer time depends on both data size and the target platform. Using the target platform to extract raw data from the target binary is effective to decrease mistakes and shorten development time. All processes in the software and the hardware design flow are time consuming, specialized, and manual tasks, even if programmers just change dataflow a slightly and test the new one. Heterogeneous platforms incorporating CPUs, GPUs, and FPGAs are the most difficult to target, as there are so many possible mappings of tasks on the different computational devices.

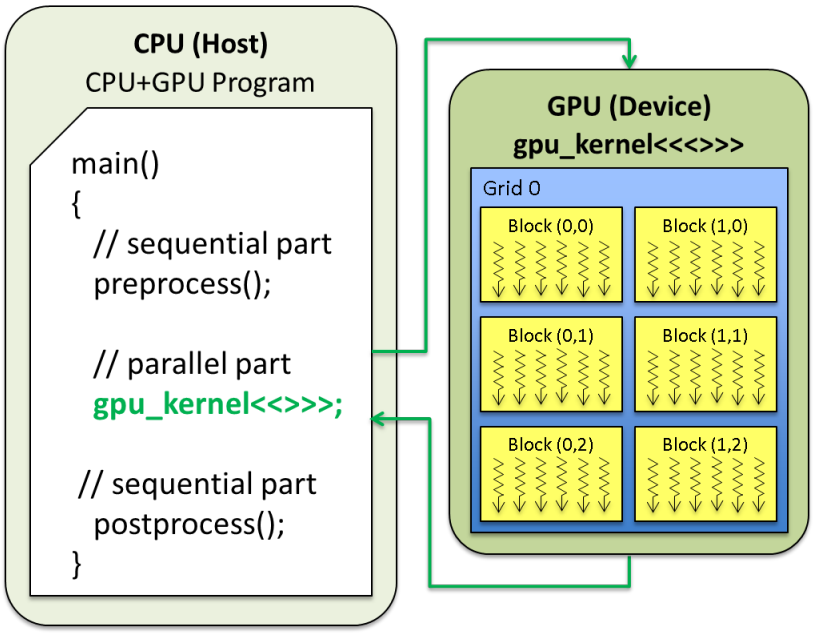


Figure 2.25: Example of CPU-GPU programming. Parallelizable parts are off-loaded to GPU.

2.4.1 Programming environment of a mixed CPU-GPU platform.

Figure 2.25 shows the processing flow of CPU-GPU programming. One of its key points is data transfer. Even if GPU shortens the off-loaded processing, if total time spent on the data transfer and processing is longer than that of CPU, it is meaningless. Reducing the number of data transfers between CPU and GPU is a critical challenge on such heterogeneous platforms. Consequently, the users should carefully choose the parts that should be off-loaded to the GPU. However, this is not easy since it often becomes a trial-and-error process and requires special knowledge [19].

2.4.2 Programming environment of a multi-node mixed CPU-GPU platform.

To communicate with another node, Message Passing Interface (MPI) is very widely used. MPI is a standardized and portable message-passing system designed by group of researchers. MPI defines the syntax and semantics of a core of library routines useful to a wide range of users. There are some implementations of MPI for high performance computers: OpenMPI [32], MVAPICH [33] or IntelMPI [34]. Although the application programming interface (function name) is almost the same, the performances of implementations are different. As I explained previously, the data must be going through the CPU's memory when the GPU wants to communicate with the GPU on another node since MPI is for CPUs. GPUs cannot handle MPI. Additionally, the total processing time including GPU processing and data transfer must be shorter than that of CPU. This is the same key point as a single-node mixed CPU-GPU platform.

Figure 2.26 illustrates an example of MPI communication on two nodes. Each node has two host CPUs and two GPUs. Host CPUs first prepare the input data and send the data to GPUs, and then

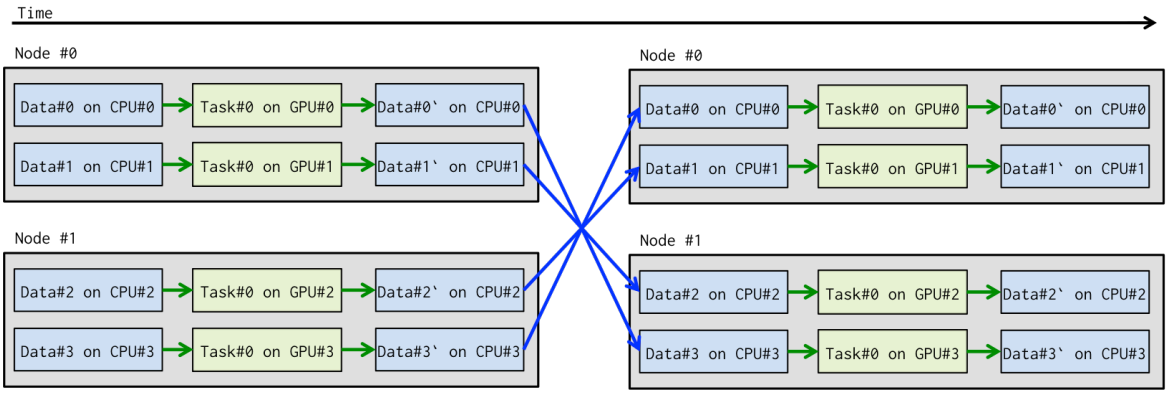


Figure 2.26: Example of MPI communication on a multi-node mixed CPU-GPU platform.

GPUs execute the same processing. When GPUs finish the processing, the result data are sent back to the CPUs. After that, CPUs send the data to another node. A series of processing is executed until the final result is obtained. The processing of a single node is almost the same as that for a single-node mixed CPU-GPU platform. This kind of processing can be seen in the scientific computing applications, such as earthquake simulation or computational fluid dynamics.

2.4.3 Programming environment of a single-node mixed CPU-FPGA platform.

From a programming perspective, Xilinx provides a software development kit for this platform, called Vivado [35]. Vivado includes all necessary components to develop and implement a solution on the platform. Programming difficulty can be alleviated by the vendor's SDK, but design complexity is still a problem. As I mentioned in Section 2.4.1, the key point of a mixed CPU and accelerator platform is data transfer. Even if FPGA shortens the off-loaded processing, if total time spent on the data transfer and processing is longer than that of CPU, it is meaningless. Thus, the users should carefully choose the parts to be off-loaded.

In the case of a mixed CPU-FPGA platform, multiple hardware modules can exist on an FPGA part. This is the main difference from a mixed CPU-GPU platform since the CPU-GPU platform rarely executes multiple functions on the GPU side. Thus, a combination of multiple functions on a CPU and multiple modules on an FPGA is a unique problem. Handling these functions and modules is not fully supported by the vendor's SDK. Additionally, not much research has been done into handling such situation.

2.5 Chapter Summary

This section overviewed three heterogeneous platforms including a mixed CPU-GPU, a multiple mixed CPU-GPU, and a mixed CPU-FPGA platform. went into detail on accelerators of each platform. On hardware side, GPU has a hierarchical memory and processing core. Cluster GPU has a communication bottleneck, and PEACH2 is introduced, which lowers communication latency among multiple nodes, to remove the bottleneck. FPGA has a mesh architecture and realizes an arbitrary circuit. On the software side, there are big differences between the platforms as well. CUDA for GPU adopts the SIMT model to control thousands of cores simultaneously. HDL for FPGA adopts a very low level programming model since it should express timing behavior accurately. Although there are many differences, these platforms share the same idea when the programmer wants to use them for accelerating the application. The idea is to "off-load" computational intensive parts on a CPU to GPU/FPGA and shorten the processing time. Common problem of these platforms are the difficulty of programming and complexity of choosing parts to be off-loaded.

Chapter 3

Courier: A Toolchain for Automatic Function Off-load on a CPU-GPU Platform

This chapter proposes 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 target platforms of this chapter are a single mixed CPU-GPU and a multiple mixed CPU-GPU platforms.

This chapter first presents Courier, including its features designed for detecting a processing flow and function off-loading. Then, the technical details of our function off-loading mechanism is describes. Then an application acceleration technique on a multiple mixed CPU-GPU platform is presented. Section 3.4 gives case study, showing the capability of Courier. I discuss our toolchain and related work in Section 3.5.

A short summary of this chapter is 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.
- It is proposed that an automatic processing flow graph generation method of analyzed functions from a running binary. The method includes tracing sub-programs to analyze functions and a heuristic approach to detect causality.
- It is proposed 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-loaded 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.

3.1 Courier: A Toolchain for Application Acceleration

3.1.1 Fundamental Concept

On designing Courier, I considered the following items. This sub section describes how I deal with these items and fundamental concept of proposed toolchain. Section 3.5 discusses the differences among related work.

I. Wider range of target users by automating acceleration processes.

II. Larger target applications by using existing accelerator functions.

III. Broader applicability to many heterogeneous platforms by adopting modularized components.

I. Wider range of target users by automating acceleration processes

This comes into consideration while referring related work. Most researches on developing work-flow for off-loading focus on a programmer who understands 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 [11], accelerator management [12] [13], and accelerator kernel implementation [14] [8] [15] [16]. Unlike them, I do not intend to generate the accelerator code itself. I 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 do not need to have the source code of the acceleration target. Our toolchain extracts the call flow of functions, and finds the part which can be off-loaded to the accelerator during execution of the binary code. The 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 not the target of the conventional work-flow.

II. Larger target applications by using existing accelerator functions

This consideration is derived from the following assumption. A data transfer time between CPU and accelerators becomes critical since state-of-the-art CPU is fast enough for calculating small data. I should focus on function-level processing flow for recent heterogeneous platforms. This is because there are many researches and commercial tools for generating function itself. A lot of recent 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 [36]. If we understand the flow of the running binary and find the parts which can be accelerated, I can automatically accelerate a target application entirely by replacing the parts with the corresponding functions of the accelerator.

III. Broader applicability to heterogeneous platforms by adopting modularized components

This consideration is similar to designing a compiler such as gcc [37] or llvm [38]. Compilers

are often highly-modularized so as to support many processors and generate better assembler code. They are composed of Frontend, Intermediate Representation (IR) and Backend [39]. In the case of C language, Frontend and IR perform the same processing such as lexical analysis or syntax tree construction. Backend performs both common optimizations and processor specific optimizations. I do not go into detail of compiler since this is not a main topic of this thesis. Structure of Courier is designed by reference to that of compilers.

3.1.2 Overview of Courier

Figure 3.1 shows the overview of Courier. It consist of *Frontend (Runtime Analyzer)* and *Backend (Function Off-loader)*. *Courier Intermediate Representation (IR)* is a bridge between them. Each role is as follows.

- Frontend automatically traces a target running binary and tries to detect a processing flow of functions by using a heuristic approach. Functions in the detected processing flow are recognized as the target of acceleration without needing access to the original source code or any sort of re-compilation. Property of input/output data is also analyzed and taken into consideration during the acceleration process.
- Courier IR represents the processing flow in a graph and code. A *task graph* is constructed to understand and find parts that should be off-loaded.
- Backend automatically performs the off-loading of the functions, if the functions uses a wide-spread function library like OpenCV, BLAS or FFT, and the corresponding function is ready for the accelerator. Dynamic Off-loader in Backend automatically adjusts data property, reduces the number of data transfer along with off-loading, and maintains original processing flow before and after off-loading.

Example of the processing step is illustrated in Figure 3.1 and caption of the figure describes the detailed work steps. Running software binary contains a function called “accum”, which obtains 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 the corresponding accelerator function “acc_accum”. The followings sub sections are go into more detail.

3.1.3 Frontend (Runtime Analyzer)

Frontend consist of three main steps to detect a raw processing flow. It 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 a profile run.

Step 1. Frontend traces functions in the running binary by using pre-defined tracing sub-programs,

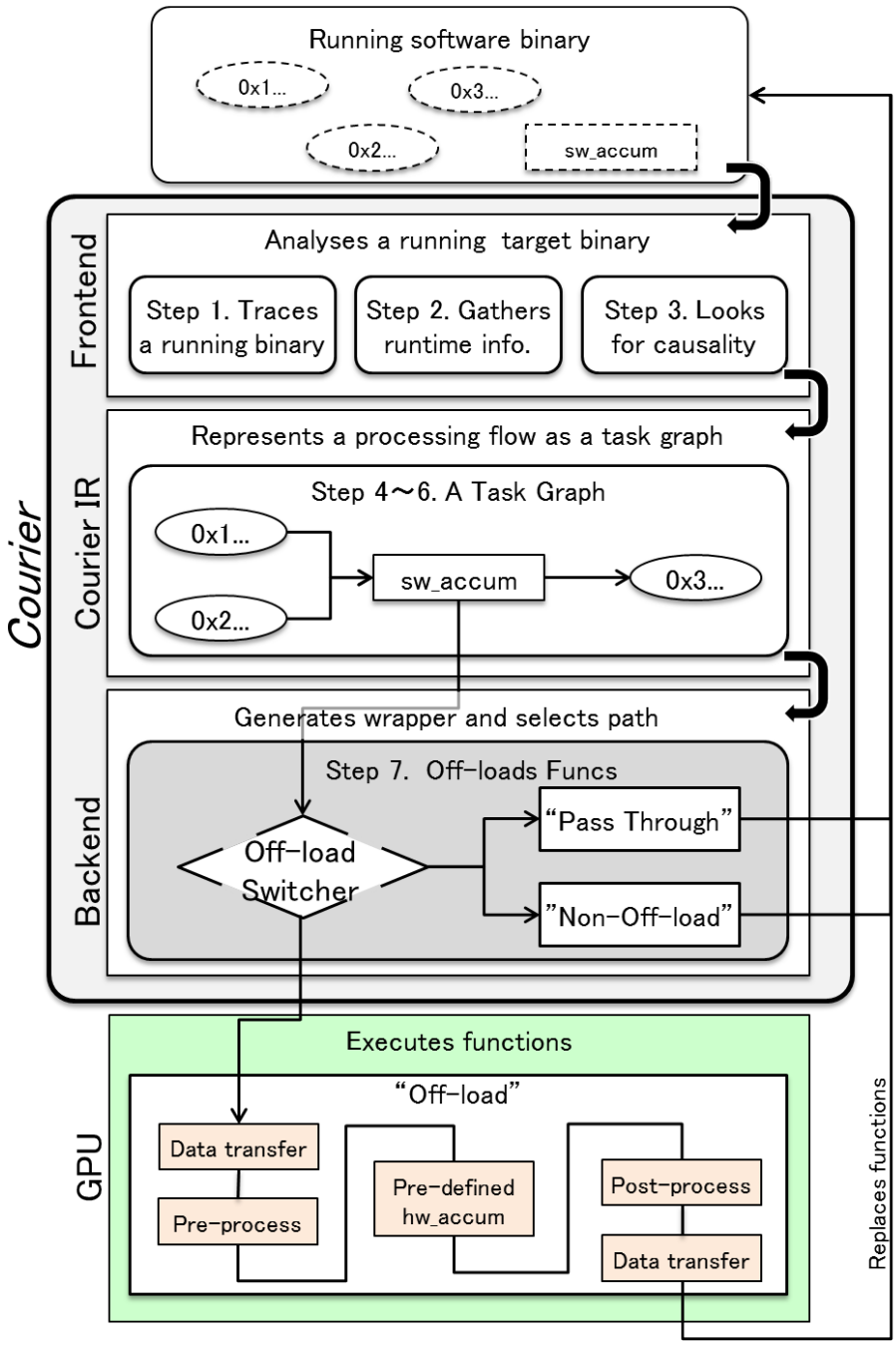


Figure 3.1: An overview and work steps of the Courier: *Frontend* traces running binary (1,2) and detects causality (3) and then generates an *Intermediate Representation* (4) and a task graph (5). Users modify off-load parts and changes IR if needed (6). Finally, *Function Off-loader* replaces function and off-loads to accelerator (7).

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 processing flow [16]. I intend to understand

processing flow and find parts that can be off-loaded on a current heterogeneous platform.

3.1.3.1 Tracing a running binary

Step 1 performs dynamic program analysis by using a tracing sub-program for each function in order to obtain runtime information from a target binary. Dynamic program analysis by using tracing sub-programs in Step 1 is a key feature, since the static program analysis needs sophisticated pointer analysis and often cannot obtain important information such as processing time, processed raw value, transfer time, etc. 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 other approaches [16] [40] [41]. In this paper, I 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 [42]. Pin is commonly used for kernel-level profile of the target program, but Frontend uses 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 conventional 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 the Courier implementation so as to improve applicability to support new libraries. By adding a new tracing sub-program for a specific library, Courier can trace the library.

```
void add(InputArray src1, InputArray src2, // input
         OutputArray dst,                  // output
         InputArray mask=noArray(), int dtype=-1);
```

Listing 3.1: Function definition of cv::add in OpenCV

Tracing sub-program for OpenCV knows the name (*cv::Sobel*), a data structure of cv::InputArray / OutputArray and the role of each argument. The role of arguments are as follows, 1st/2nd are input/output data, 3rd is the depth of output data, and 4th/5th are x-/y-axis parameters. Frontend obtains the raw value in the running binary by using the information at entry and return point.

Figure 3.2 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

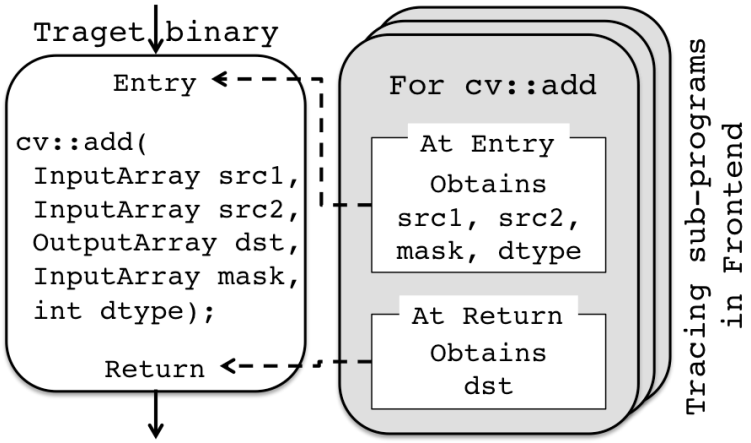


Figure 3.2: Tracing sub-programs for specific functions in Frontend access designated arguments and obtain raw values at entry and return points of the functions.

```
[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 3.2: Gathered runtime information of cv::add.

(output data). Absolute times are also recorded. List 3.2 is gathered runtime information of cv::add. It's just an enumeration of the information and hidden from the users.

3.1.3.2 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. 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.

Other problems Frontend must address are application control flow, data movement, and data modification outside functions known to Frontend. Control flow can be extracted such as if-else branch and count frequency of a taken branch [40] [16]. If the branch exists, constructed graph is split into several paths. Data movement and modification outside functions can be traced, even if the target running binary did not use a library function to modify images such as pointer access. Currently, I understand both problems, but how I treat them is future work.

3.1.4 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. These steps correspond to Step 4 6 in the Figure 3.1

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 List 3.2 is translated into a more user-friendly description as shown in List 3.3. 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 processing flow. *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. Section 3.4.1 shows an example of HOG feature detection. The grammar of IR deliberately restricts the number of arguments that functions can accept.

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

Listing 3.3: Courier IR description of cv::add.

3.1.5 Backend (Function Off-loader)

Backend is designed for automatic off-load without user intervention, and consist of the following main step. These steps correspond to Step 7 in the Figure 3.1

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 I do not attempt any sort of automatic binary translation, since state-of-the-art translation techniques cannot be used here. Function Off-loader generates wrapper, replaces functions, sends data while maintaining the original processing flow and reducing the number of data transfer. Section 3.4.1 explains the details with an example of OpenCV.

3.1.6 Applicability of Courier

When the users want to support a new library, they should introduce 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 libraries without developing a new version of Courier.

A limitation of the current Courier concerns functions within control statement (e.g. if-else branch or loop). Although Courier can off-load these functions, it currently deals with the straight forward function calls in binaries in default. There are two major problems to automatically off-load such functions. One is how to detect such processing flow by Frontend and the other is how to off-load the function effectively. In the case of functions inside a loop, Frontend cannot recognize that they are inside a loop and just called iteratively. If the number of iteration of a loop is fixed, off-load can be done effectively by using the current version of Courier. On the other hand, it requires another method to off-load non-fixed loop effectively and this is a future work. When multiple function calls appear continuously in the target binary, Courier off-loads all of them so that the number of data transfer is reduced even if additional statements exist before and after functions. In this case, the performance can be much improved. If the corresponding functions are ready in the accelerator, any type of function calls can be off-loaded in the same manner. For example a binary program using both OpenCV and BLAS can be off-loaded.

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 exist 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.

3.2 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 OpenCV's GPU functions, cuBLAS and cuFFT are used in Section 3.4.

Many existing applications use widespread function libraries like OpenCV or BLAS. Enough optimized codes of corresponding functions are available for popular accelerators. Courier is a powerful tool for users who want to accelerate a current running application with such a library using accelerators without knowledge about the source code.

3.2.1 Dynamic Linking and Its Problems

Function Off-loader adopts dynamic linking mechanism on Linux environment 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 deployment. Although this technique is known as DLL hijacking or DLL injection, here, the purpose is off-loading and some problems occur.

There are three main problems that occur if we just use DLL injection for the function off-loading. The first is unconditional off-loading (Figure 3.3, UNCOND-OFF), the second is a restriction of the number of inputs/output of substitute function (Figure 3.4, SAME-INOUT), and the third is redundant data transfer when a series of functions is off-loaded (Figure 3.5, 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 Figure 3.3, 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

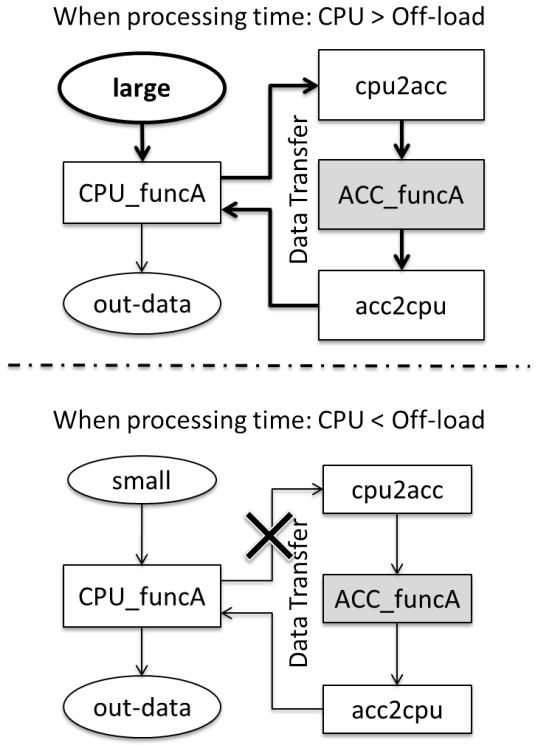


Figure 3.3: UNCOND-OFF: Typical dynamic linking replaces all the same name functions in the target binary.

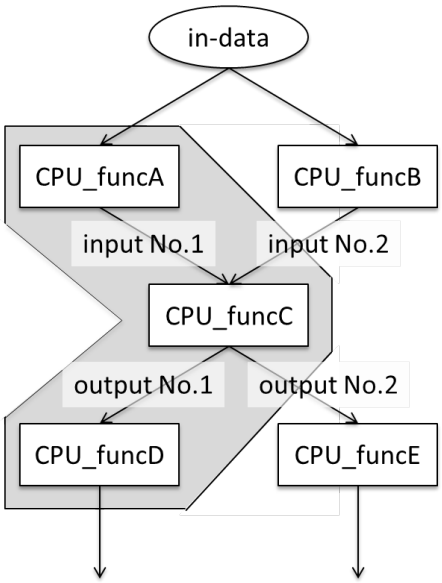


Figure 3.4: SAME-INOUT: Corresponding function must have the same number of input/output. Two input/output functions can only be the corresponding.

restriction. In the case of Figure 3.4, the colored part has two input and two output, and "CPU_funcA" has one inputs and one outputs. "CPU_funcA" cannot be replaced with the colored part since the number of input/output are the different. I'm 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, *Function Off-loader* and *Off-load Switcher* are introduced.

3.2.2 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 3.7 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 the 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.

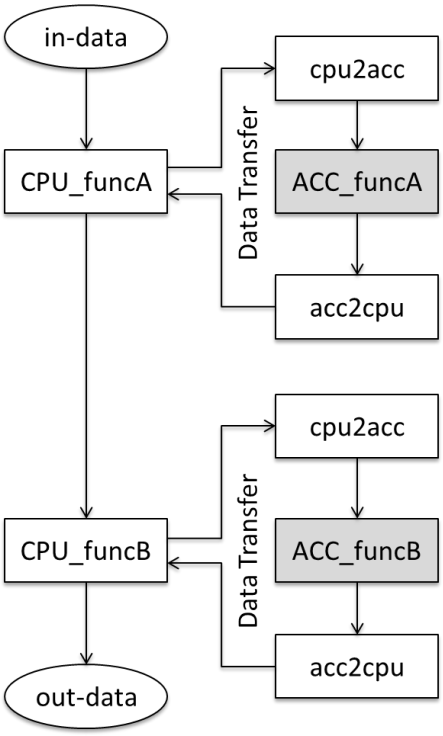


Figure 3.5: RDNT-TXRX: Redundant data transfer happens when a series of functions are off-loaded.

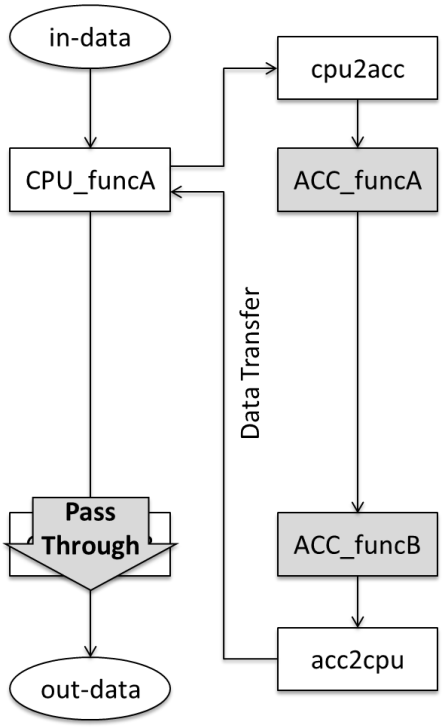


Figure 3.6: *Function Off-loader* reduces the number of data transfer and maintains the original processing flow.

To deal with the above described UNCOND-OFF and RDNT-TXRX problems, *Off-load switcher* is introduced 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 [43], which are APIs for dynamic loading in Linux. In Figure 3.7, *Off-load switcher* is shown at the top, and the three paths work as follows.

- *Non-off-load* executes the original function, 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. Figures 3.5 and 3.6 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

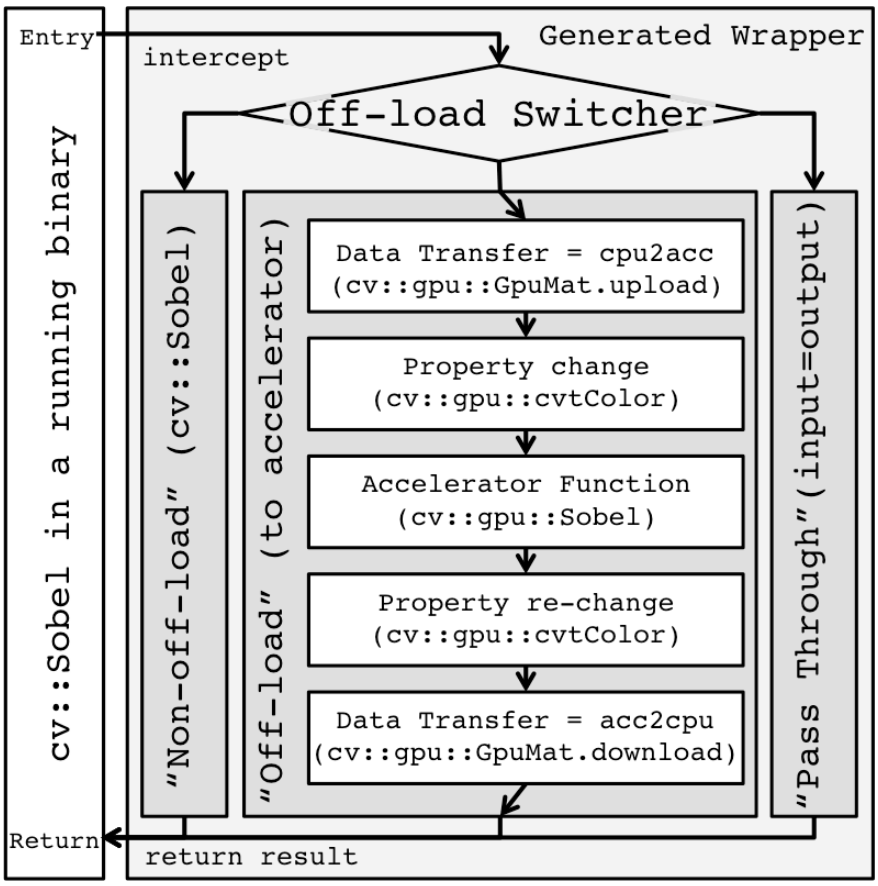


Figure 3.7: Function Off-loader and generated wrapper for OpenCV: One of three paths is selected: *"Non-off-load"*, *"off-load"* and *"Pass Through"*

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

3.3 Task Pipelining on Multiple GPU Platform

In multiple GPUs environment such as Tsubame supercomputer, data level parallelism (DLP) is typically used to implement application [44]. On the other hand, task level pipelining on multiple GPUs is not mainstream. In this section, multiple GPUs are directly connected as a form of a pipeline computation by using PEACH2. Here, I call it *Inter Accelerator Pipelining (IAP)* as a special case of task level pipeline. This section describes a concept and an implementation of IAP on TCA.

3.3.1 Basic Idea

IAP connects multiple accelerators (e.g. GPU or FPGA) as a form of a pipeline computation. An application is divided into some tasks, each of which is assigned to an accelerator. Task means larger process such as API-function-level, not "fine grained" such as x86 assembly [45]. IAP suits for stream computation such as image processing or computational fluid dynamics (CFD). I call this type of implementation "IAP style".

Compared to DLP, IAP doesn't increase required memory bandwidth along with the number of devices. In the case of DLP which is shown in Figure 3.8, required memory bandwidth linearly increases, since different source data are required for each device. At first, the master host CPU sends the data to all host CPUs, then each host CPU sends the data to each GPU, and then each GPU starts processing. Each processing step becomes sequential and scatter/gather process is required along with each step as well. In the case of IAP on ordinary multiple mixed CPU-GPU nodes, which is shown in Figure 3.9, the host CPU sends the data to GPU in the node, then GPU starts processing. Finally, result data are sent to the next pipeline stage on another GPU via host CPU. Communication via host CPU still degrades performance. This is one of the reasons why task level pipeline on multiple GPUs is not easy.

Compared to IAP on ordinary multiple mixed CPU-GPU nodes, IAP on TCA nodes can eliminate extra copy to CPU. Figure 3.10 illustrates the implementation. In the case of IAP on TCA which is described in Section 2.2, direct communication is enabled thanks to PEACH2, and communication latency is dramatically reduced. From the point of view of the pipeline, reducing the communication latency corresponds to shortening the processing time of each stage. As a result, the whole processing time of a pipeline is reduced. Additionally, other benefits can be found when we implement stream application. First, merging output data from multiple GPUs into one large output data is done in chronological order. The pipeline gets input data and generates output data in chronological order naturally. On the other hand, when DLP style is used, extra merging process is needed. Applications are implemented in task level pipeline manner on multiple TCA nodes so as to explore the capability of pipeline implementation on multiple GPUs.

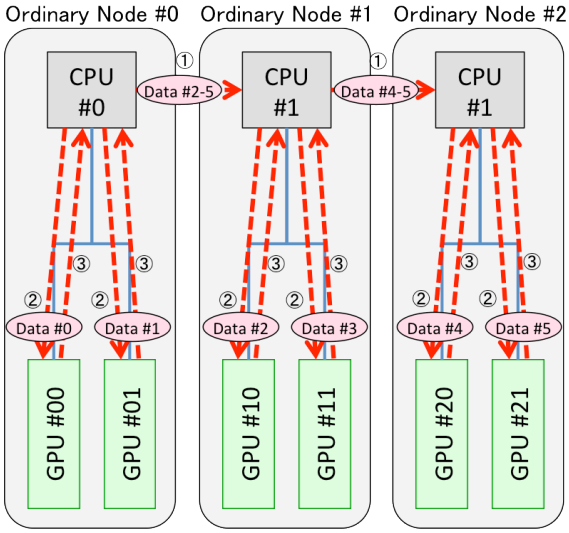


Figure 3.8: Implementation of data level parallel. CPU runs one or two threads to control GPUs and each GPU runs the same processing. (DLP1, DLP2)

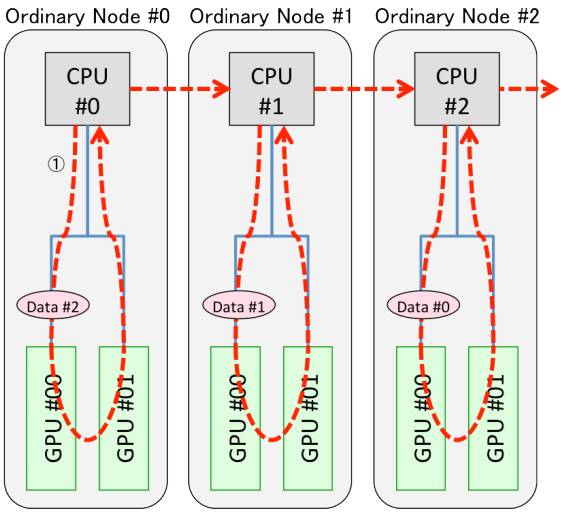


Figure 3.9: Implementation of inter-accelerator pipeline without PEACH2. Data transfer time between GPUs degrade total performance. (IAP1, IAP4)

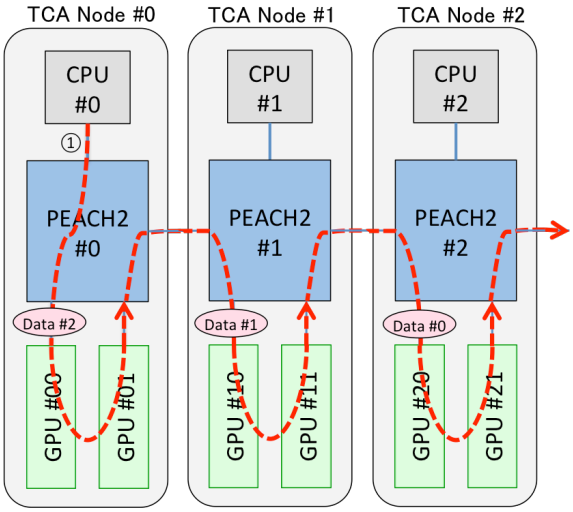


Figure 3.10: Implementation of inter-accelerator pipeline with PEACH2. Stage tokens and data are sent to CPU and PEACH2 respectively. (IAP4-P2)

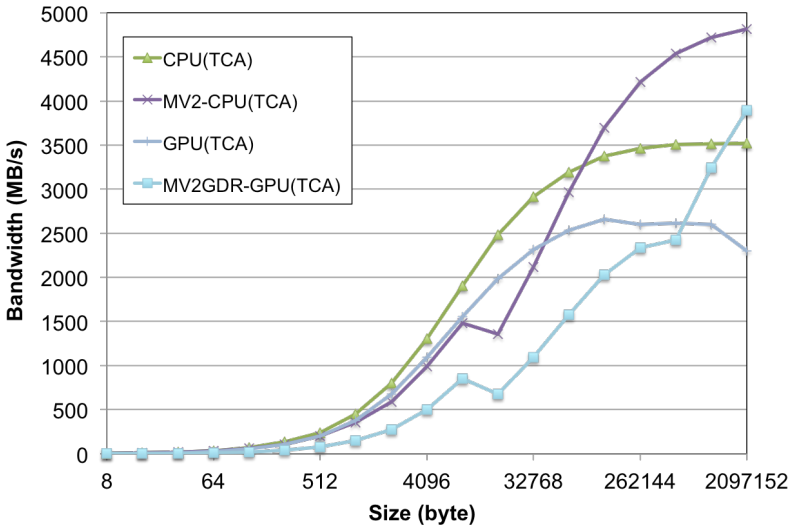


Figure 3.11: Bandwidth between CPU/GPUs in different nodes: PEACH2 reaches to 93% of theoretical peak performance of PCIe. At 8KB, Bandwidth reaches 1.6GB/s and 4 time faster than MVAPICH2.

3.3.2 Exploratory Evaluation of TCA

It is conducted that an exploratory evaluation of communication latency and bandwidth of TCA. Figure 3.11 and Figure 3.12 show communication latency and bandwidth between GPUs and CPU over different nodes. The following is a legend of lines.

- CPU(TCA) shows a CPU to CPU communication between two TCA nodes via PEACH2.
- GPU(TCA) shows a GPU to GPU direct communication between two TCA nodes via PEACH2.
- MV2-CPU(TCA) shows a CPU to CPU communication between two TCA nodes via MVAPICH2.
- MV2GDR-GPU(TCA) shows a GPU to GPU direct communication between two TCA nodes via MVAPICH2's GPU direct RDMA feature.
- MV2-GPU(SB) shows a GPU to GPU indirect communication between two nodes which have Intel's Sandy Bridge CPU via MVAPICH2.

Average latency between two GPUs is around 2.5 μsec when the data size is less than 2048 bytes. It is around 3 times faster than MVAPICH2-GDR, and around 8 times faster than MVAPICH2. Note that, MVAPICH2-GDR uses *cudaMemcpy()* on Unified Virtual Address (UVA) environment, and MVAPICH2 uses *mpi_send()*. About bandwidth, 93% of theoretical peak performance is achieved when the data size is 256 KB. In particular, PEACH2 provides better performance over MVAPICH2 both in latency and bandwidth.

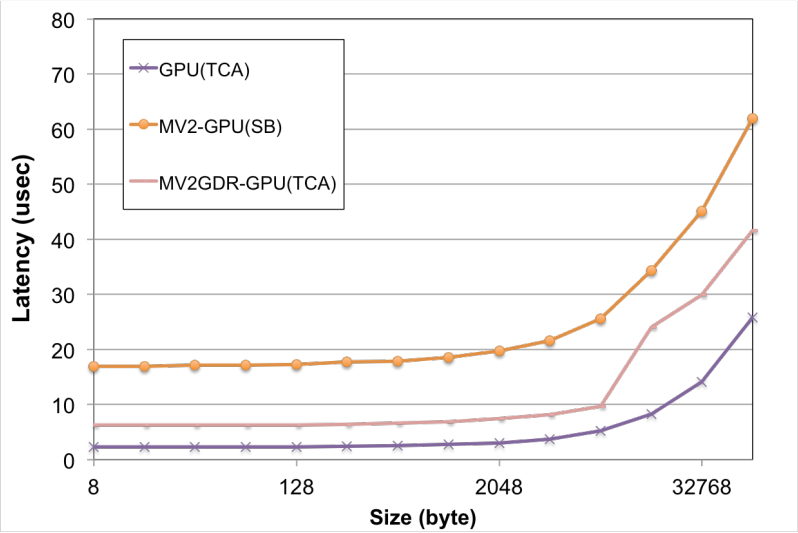


Figure 3.12: Latency between GPUs in different nodes: PEACH2 achieves 1/7.3 of that of MVAPICH2, and 1/2.7 of that of MVAPICH2-GDR. PEACH2 achieves lowest latency in all parts.

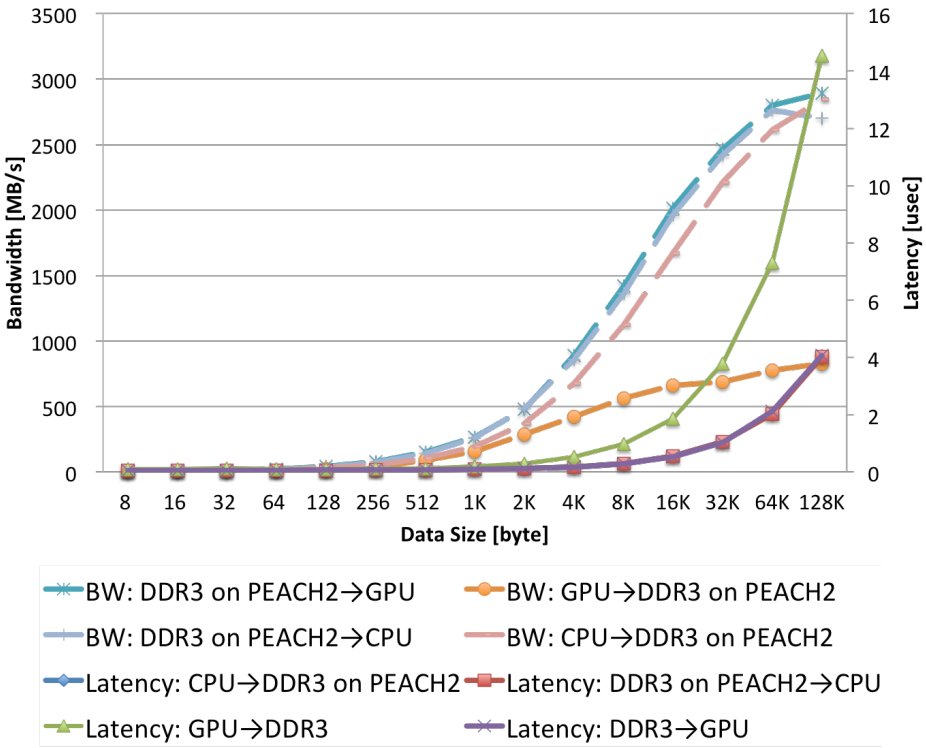


Figure 3.13: Latency (left axis, solid line) and Bandwidth (right axis, dash line) between CPU/GPU-DDR3 on PEACH2: Except for GPU to DDR3 on PEACH2, latency is less than 4 μsec. Bandwidth reaches to 78% of theoretical peak performance.

The latency and bandwidth of DDR3 on PEACH2 memory is also shown in Figure 3.13. Latency is less than 4μsec in all parts except for GPU to DDR3 on PEACH2. Besides, bandwidth reaches 78% of theoretical peak performance and 4.8Gbps when the data size is 128 KB. In particular, DDR3

on PEACH2 memory can be used when the data size is large. In Section 3.4, application transfers the image data that the size is larger than 2MB. Note that latency and bandwidth from GPU to DDR3 on PEACH2 are much worse than the others. This is caused by smallness of PCIe buffer of Intel's Sandy Bridge architecture. This problem will be solved on Intel's Ivy Bridge architecture.

There are some researches on direct communication network system. APEnet+ [24] is one of the closest idea to PEACH2. Unlike PEACH2 which is compatible with PCIe protocol, APEnet+ uses their own protocol to realize direct connection between GPUs and network interface. PEACH2 has an advantage to APENet+ since it isn't required to convert protocols. The CUDA 5 programming environment for NVIDIA GPU provides the RDMA mechanism to the GPU memory, called GPUDirect Support for RDMA, with a Kepler-class GPU [20]. This mechanism enables zero-copy communication between the InfiniBand Host Adapter and GPUs [25]. Although PEACH2 also uses the same RDMA mechanism, PEACH2 can eliminate the overhead for protocol conversion from PCIe to the InfiniBand packet, and the overhead of MPI protocol stack.

3.3.3 Implementation of intra-node pipeline

Figure 3.14 depicts an implementation and pipeline overview of task pipeline using Intel Threading Building Blocks (Intel TBB) and OpenMPI on ordinary multiple mixed CPU-GPU nodes. Blue line shows task token communication and green line shows CUDA based data transfer. Both token and data are sent to the master thread in host CPU. Figure 3.15 also depicts that on TCA nodes. Orange line shows inter-node communication via PEACH2. PEACH2 has its own communication API and I used it.

Implementation of task level pipeline on multiple GPUs is not easy since it requires many parallel programming techniques such as inter node pipeline, intra-node pipeline, and order of inter/intra-node task stages of pipeline. Furthermore, to build an efficient inter accelerator pipeline on multiple GPUs, three major problems remain. First is pipeline execution itself, the second is communication between accelerators, and the third is storage which stores inter-accelerator data. It is used that Intel Thread Building Blocks and OpenMPI for the first problem. And PEACH2 is used to address the remaining two problems.

3.3.3.1 Building the intra-node pipeline

There are some open source libraries which realize task pipeline on CPU, for example pthreads [46], Boost::thread [47], OpenMP [48], and Glib::thread [49]. OpenMP requires tricky programming or extension to realize pipeline execution [50] [51], since it is intended for exploiting loop level parallelism. The others are also developed for exploiting thread level parallelism, not task level parallelism or pipeline.

TBB is flexible open source library for parallel execution. It provides software controlled task pipeline using multiple threads. The *tbb::pipeline* class is provided to implement task pipeline, and

each stage has a token to keep the order of pipeline. A user first writes each task which runs in each stage of the pipeline. Then the user registers tasks and also defines a processing order of tasks to TBB. In the Section 3.4, an image processing algorithm called Sobel operator are divided into three tasks, and process them in a simple straight forward pipelined manner.

In TBB, data and task assignment to threads is controlled by a master thread. Besides, each task runs on the slave thread. This type of pipeline makes it easy to reorder tasks and insert new tasks. Input/output data to/from tasks and storing are done when the master thread receives the finish token from slave threads. TBB 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 the finish token to the master thread. TBB can deal with tasks which are written in CUDA. To our knowledge, the other libraries are more difficult than TBB to make task pipeline and GPU task. TBB also realizes a double buffering when two or more GPU tasks are registered. The detail of pipeline structure of TBB is described in the reference [52].

It is conducted that a preliminary evaluation of scalability of TBB pipeline. The target application was applying 100 input images to Sobel operator 32 times. For it, 32 stages straight forward pipeline is implemented. The evaluation is done on TCA node and a CPU. When 13 stages run in parallel, it achieved 89% speed up compared to a single stage. Additionally, when 32 stages run in parallel, they achieved 92% speed up while the speed up is saturated. According to the evaluation, TBB can construct effective task pipeline. This preliminary evaluation demonstrates that Intel TBB is effective to construct the task level pipeline.

3.3.3.2 Building an inter-node pipeline

Blocking communication is required to build an inter-node pipeline, since the first task stage of each node has to start its process after input data from previous node arrives. Furthermore, the combination of Intel TBB and blocking communication (OpenMPI or PEACH2 API) naturally maintains the order of the whole task stage. Task level pipeline on multiple node can be built.

In the case of ordinary node (Figure 3.14), OpenMPI is used to distribute program, communicate data, and synchronize multiple nodes. *MPI_Send* and *MPI_Recv* are used to blocking communication. *MPI_Isend* and *MPI_Irecv* can be used for non-blocking communication, but it collapsed the processing order of pipeline.

In the case of TCA (Figure 3.15), PEACH2 API is used to communicate data and synchronize multiple nodes. OpenMPI is used just to distribute program. Similar to Figure 3.15, PEACH2's blocking communication is used. Polling of special register is required for current version of PEACH2 API. Once after data communication finished, the sender task sets a finish flag to a special register. The receiver task polls the register and then starts own process.

3.3.3.3 Storing inter-task data to arbitrary memory

By default, a task of TBB pipeline returns inter-task data to the master thread, and also returns a token. Redundant copy, non-direct GPU communication in the same node and that of OpenMPI in the different node, occurs even if TCA is used. To store inter-task data to arbitrary memory, each task of TBB is modified. TBB's pipeline (tbb::pipeline class) uses simple token, which is illustrated in blue line in Figure 3.15, to maintain processing flow. Each TBB task is changed not to return the inter-task data as follows. After finishing the processing, each task sends just the property (address, size and bit depth) of the data instead of the data itself. In the case of intra-node, the next task receives data based on this address via direct GPU communication (*cudaMemcpyPeer*). In the case of inter-node, the next task in the next node receives data via PEACH2 API or OpenMPI. Currently, the property is just eight bytes. In the case of ordinary node (Figure 3.14), OpenMPI is used to send/receive data to CPU in the next node and redundant copy is unavoidable. On the other hand, in the case of TCA (Figure 3.15), PEACH2 API is used to communicate data. A GPU in the next node can directly receive data, and redundant copy is completely eliminated.

Enabling to use arbitrary memory is efficient. It eliminates redundant copy and useful for storing large size data. Inter-accelerator pipeline often produces large size inter-task data between tasks like the task level pipeline. For example, an FIFO is a common way to store such inter task data to keep the processing order. Once processing time becomes imbalanced, the FIFO depth increases and large size memory is also required. Without the technique that is described here, multiple redundant copy easily degrades performance.

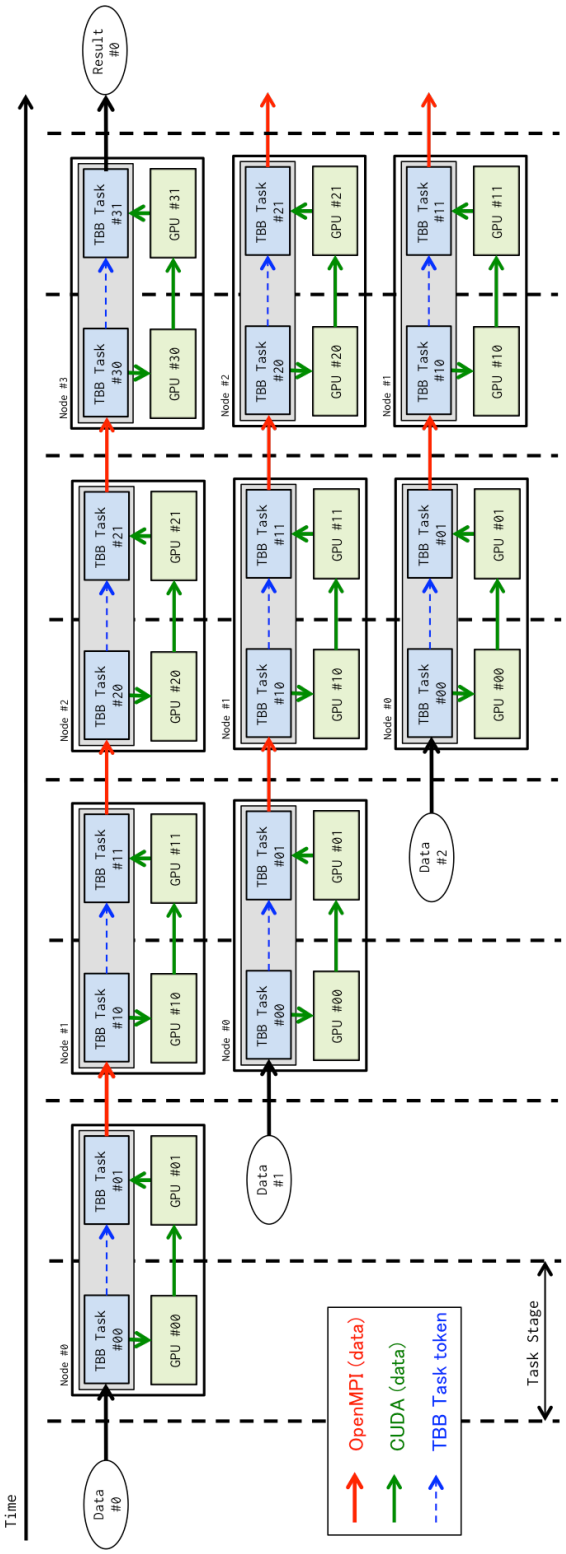


Figure 3.14: Implementation and pipeline overview of inter-accelerator pipeline without PEACH2. Role of TBB tasks on CPU is inter-node pipeline which includes send/receive data with OpenMPI, and control pipeline execution of two GPUs. OpenMPI distributes program to each node as well. Data transfer time between GPUs degrade total performance. (IAP1, IAP4)

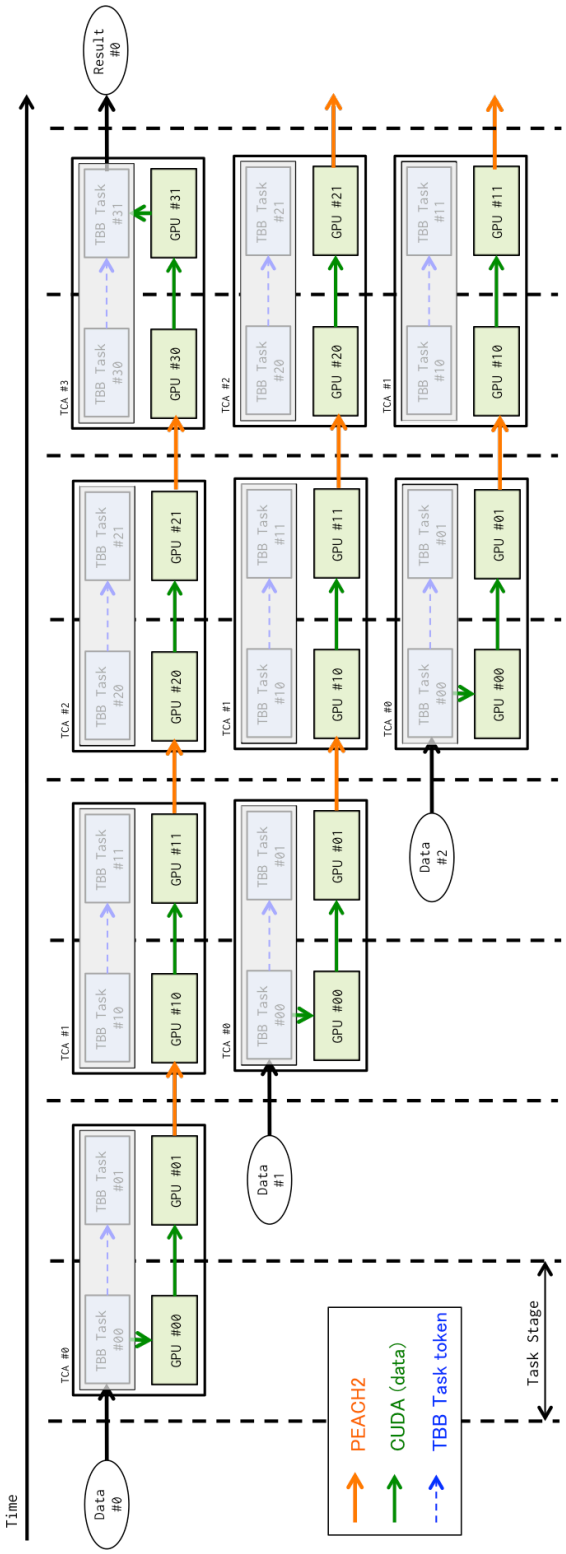


Figure 3.15: Implementation and pipeline overview of inter-accelerator pipeline with PEACH2. Role of TBB tasks on CPU is just control two GPUs, and kick PEACH2's DMA controller. Then PEACH2 sends the data to next node. OpenMPI distributes program to each node. Redundant copies are deleted and data movement becomes simple. (IAP4-P2)

3.4 Case Study

This section shows four case studies. First three case studies illustrate work-flow of Courier on a single CPU-GPU platform. Three case studies are 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 Fourier transform (FFT). Then this section shows an case study of Inter Accelerator Pipelining (IAP) on TCA node. Simple image processing application is implemented in IAP style, and execution time is evaluated.

Environments of the first three case studies is as follows. Host OS is Fedora 20 64bit (kernel 3.14.3), CPU is Intel Core i7-3770K 3.5GHz, and the accelerator is NVIDIA GeForce GTX670 with PCIe Gen.3. Binaries are compiled with GCC ver.4.7. Environments of the last case study is as follows. Host OS is Scientific Linux 6.3 (kernel 2.6.32), CPU is Intel Xeon E5-2670 2.6GHz, and the accelerator is NVIDIA Tesla K20m with PCIe Gen.3. PEACH2 board is installed on PCIe Gen.2 x8 slot.

3.4.1 Histogram of Oriented Gradients (HOG) on a single mixed CPU-GPU platform

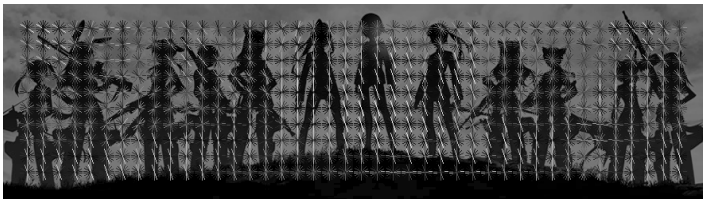
HOG is a widely used algorithm for feature detection, such as face recognition [53]. HOG application includes three main features that are commonly seen in computer vision applications: OpenCV C++ API functions, diverging/converging flow, and image duplication. Note that OpenCV functions often perform computationally intensive image transformations, and have clearly defined input and output images, and so are ideal candidates for off-loading to accelerators. The processing flow in running binary consisted of the following three main steps. Step 1) Compute gradient and magnitude, Step 2) Gradient adjustment, and Step 3) Create histogram. Pseudo code shown in List 3.4 is just for explanation since neither Courier nor the user needs access to the source code.

1. Compute gradient and magnitude: Each frame of the video source is converted into gray scale by *cv::cvtColor* and *cv::split*. Then, x and y axis Sobel operators (*cv::Sobel_x, cv::Sobel_y*) are applied. Both operators obtain the same gray scale image. The gradient and magnitude are calculated from the x/y Sobel images by using *cv::cartToPolar*.

2. Gradient adjustment: gradient values are adjusted to within 0 to 180 degrees by *cv::threshold* and *cv::subtract*. $if(a > 180)\ \ a = a - 180, \quad if(a < 180)\ \ a = 0$

3. Create histogram: The two images generated in Step 2 are combined into one image (*cv::add*), and adjusted gradient values are calculated. Lastly, this image is divided into a nine-channel histogram by using *cv::divide*.

(a) The original image



(b) Visualized Histogram Oriented Gradient

(c) Detected people

Figure 3.16: An intermediate results of HOG.

3.4.1.1 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 Figure 3.1. 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).

These information are extracted whenever the target application calls an OpenCV function. All the OpenCV functions have the prefix "cv::", such as *cv::Sobel*. The input/output image data are actual image value. Data movement and modification outside OpenCV can be traced technically, even if the target running binary didn't use OpenCV function to modify images such as pointer access. Function type denotes one of three transform types (sink, unary, and binary), corresponding to the FunctionType variable in Courier IR.

```
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 3.4: Pseudo code of HOG feature detection in the target 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 3.5: Generated IR description of the target running binary.

II. IR Description The Courier IR description as shown in List 3.5 was automatically generated. Users modify this to change processing flow if needed (Step 6 in the Figure 3.1). At line 9, the *cv::VideoCapture* function was used to provide input images for the processing flow. Note that in the

actual software, *cv::VideoCapture* was given extra arguments which were captured in the Courier IR. However, Courier IR hid these from the user for simplicity. Furthermore, images were taken as an argument of *volatileInput* at lines 3 to 7, and so it was protected from modification since this is the very first data. Two *cv::Sobel* with different arguments were at lines 12 and 14. At line 10, *cv::Sobel* with "X-axis" argument applied a x-axis filter and the other applied a Y-axis filter. Such differences can be detected by a dynamic program profile on Frontend.

Number of input and output were correctly analyzed as well. *cv::cartToPolar*, *cv::subtract*, *cv::add*, and *cv::divide* were binary transforms, which obtained two inputs and generated one output. All the other functions were unary transforms, which have one input/output, and the column order corresponds to the original processing order.

III. Generating a task graph After the profile run, a task graph of the binary was automatically generated, which is shown on the left of Figure 3.17. User can refer the graph and decide off-load and non-off-load parts if needed (Step 6 in Figure 3.1). The graph was 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 was 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 7.91Mbit (1920×1080×32bit×1-channel) and 1.98Mbit (the same property, but 8bit), respectively. Dynamic program analysis on Frontend correctly extracts the runtime information.

The graph also illustrates that this binary included typical branching and converging, for example both *cv::Sobel* operators used the same image as an input, and these output images became inputs of *cv::cartToPolar*. Furthermore, the vertical relative offsets (separated by dash lines) illustrate sequential execution, which is an opportunity to exploit function level parallelism. Additionally, *cv::Sobel_y* created 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 Figure 3.1), Courier searched for "safely off-loadable" parts, and found that all of the functions are candidates. Courier automatically off-loaded them by using Function Off-loader and existing corresponding library. Finally, Courier updated the IR and introduces the following new lines at the end of List 3.5: *cpu2acc* intercepted *cv::cvtColor* function when it was called in running binary. *acc2cpu* sent back "gradient" image date to the original binary.

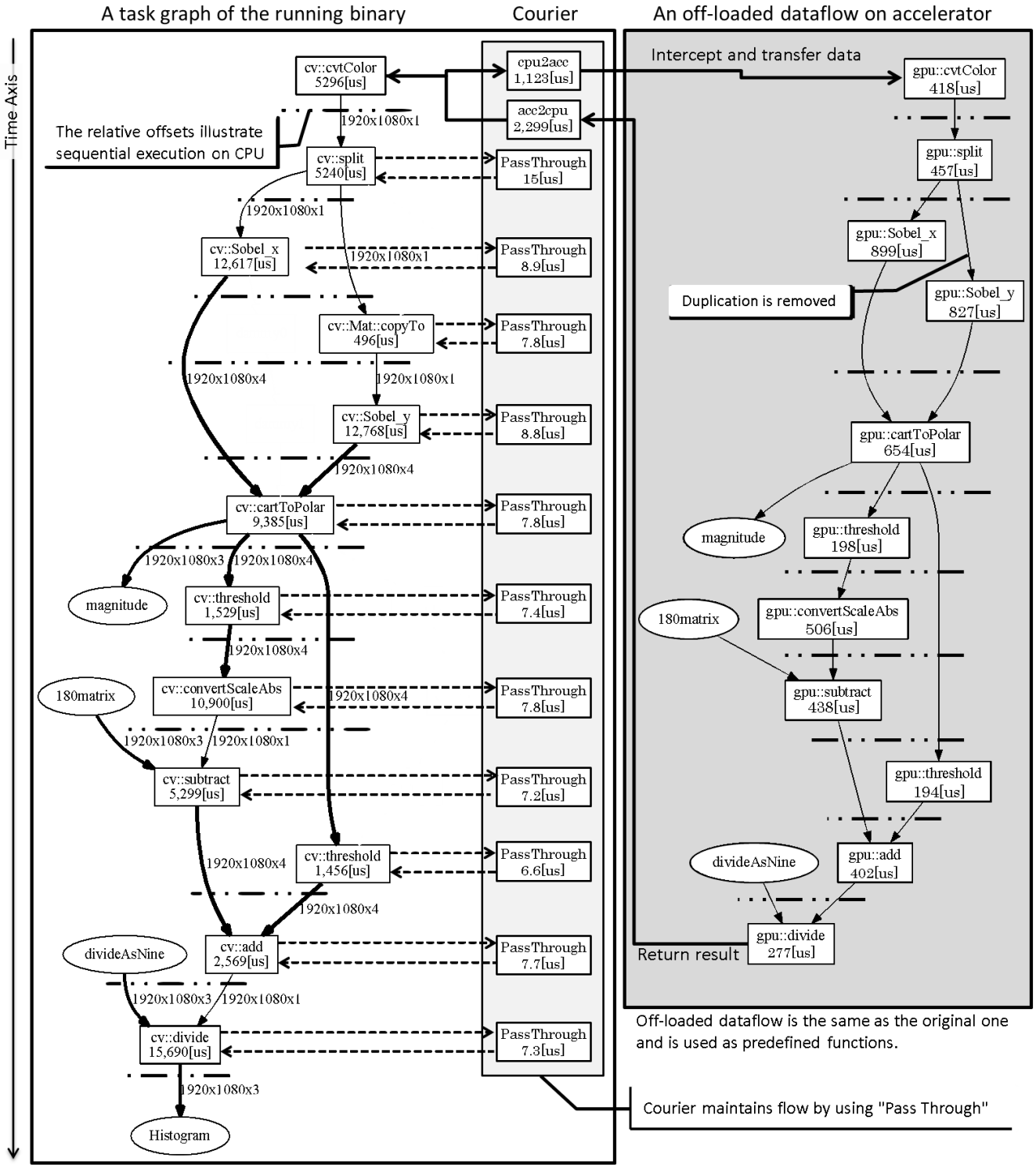


Figure 3.17: Generated task graph from running binary of HOG (left) and off-loaded functions (right) with notations. Function Off-loader generated the wrapper for the functions within *cpu2acc* and *acc2cpu*. It also selected the path of "Off-load" and maintained the processing flow by using "Pass Through".

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

Listing 3.6: Additional statements to off-load functions

In Courier for OpenCV, *cpu2acc* copied and transferred designated images to the accelerator. The following images and functions run on the accelerator until *acc2cpu* function was called.

Courier also selected the "Pass Through" pass in Function Off-loader to reduce the number of data transfer and maintain the original processing flow. (to deal with a RDNT-RXTX problem which is shown in Figure 3.6.) For the first purpose, Function Off-loader intercepted *cv::cvtColor* as "the head" of a series of functions and "off-loads", and then all functions were run on GPU. For the rest of functions, Courier intercepted and "Pass Through" even if a function doesn't off-loaded.

If Courier used ordinary DLL injection, nine data transfers happen. Small images (1.98MB) took 447[μs] to make a round trip, while large ones (7.91MB) took 3,176[μs], respectively. Nine data transfers (two round trips for small images, and seven for large ones: $447\times2+3, 176\times7 = 23, 106$[$\mu$s]) were reduced to one (send small image and send back large one: $1, 109 + 724 = 1, 833$[μs]). Data transfer time was reduced to less than 10%, and RDNT-RXTX problem was solved.

Additional tweaks can be performed by the user. According to Figure 3.17, *cv::Mat::copyTo* seemed redundant. Therefore, this copying function could be deleted as below by the user, and Function Off-loader replaced this function with "Pass Through". Because this deletion may affect the processing flow, Courier does not automatically perform it. For such "unsafe" processing flow 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 3.7: Delete redundant copyTo

In the case of image processing, processing time can be shorten by changing processing parameters while the final result is almost the same. For example, Sobel filter used in the case study adopted 3x3 processing window, and the size of window can be changed to 5x5. This kind of modification is classified in the "unsafe" off-load by using domain specific knowledge. Although it can be realized by changing the correspondence relationship of functions, a quantitative analysis is future work.

V. Results The right side of Figure 3.17 shows the off-loaded result. Courier replaced the designated functions and maintains the original processing flow by selecting "Pass Through". On the GPU side, off-loaded version was the same as the original one and predefined accelerated functions with wrapper were used. "copyTo" did not run on GPU anymore, and was "Passed Through" in the binary.

That is, "copyTo" was deleted.

Table 3.1: 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
convertScaleAbs	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

Table 3.1 shows processing times. Courier shortened the processing time to 8,766[μs] and sped up x8.89 compared with the original binary. In Table 3.1, "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 was equal to "Off-loaded" functions. It shows that cvtColor was replaced by Courier and all functions were executed here. Additionally, processing time of acc2cpu in "Off-loaded" functions was longer than that of "Manual imple." because current acc2cpu for OpenCV included additional data copy. The data copy was required in order to assign the result data forcibly from GPU to the binary. "Manual imple." was a manually implemented GPU version of the original application. The difference between "Courier's result and "Manual imple." arose from *cpu2acc* and *acc2cpu*. Both commands included data type conversion along with data transfer from GPU to CPU. Additionally, the number of data transfer of "Courier's" result was the same as that of "Manual imple." since Function Off-loader reduced the redundant data transfer. An overhead of Function Off-loader for OpenCV was also measured . We subtracted the processing time of "Not-off-load" from ordinary run to measure an overhead of wrapper function and dynamic linking. It was around 150[μs] for each and was attributed to OpenCV data type conversion and function pointer replacing.

The former one was less than 20 [μs] and the latter one was around 130[μs]. For the "Pass Through", the overhead was around 7[μs] for each.

3.4.2 General Matrix Multiplication (GEMM) on a single mixed CPU-GPU platform

General matrix multiplication (gemm) performs the following equation $C = \alpha AB + \beta C$, and a very widely used in computational science. *dgemm* is a double precision version of gemm. We prepared binary that only had *dgemm* function of BLAS, widely used for common linear algebra operations. Binary included *cblas_dgemm* in ATLAS 3.8.4 [54] for CPU, and Courier used *cublasDgemm* in cuBLAS [5] in CUDA 5.5 for GPU.

I. Analyzing running binary We prepared a tracing sub-program for BLAS and it extracted 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 showed that all the data were protected from modification via *volatileInput/Output*.

```
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 3.8: dgemm processing flow in Courier IR.

III. Generating a task graph After the profile run, a task graph was generated which is on the left of Figure 3.18. According to the graph, *cblas_dgemm* was a ternary function, which obtained three data ("src0","src1" and "src2") and produced result data "dst". The matrices were all 2048×2048, and consequently all the ellipse nodes were the same size. CPU took 1379[ms] to process each matrices.

IV. Acceleration with Courier Courier found *cblas_dgemm* was "safely off-loadable" because all the input/output data were extracted, and the corresponding accelerator function *cublasDgemm* was available. Then *cblas_dgemm* was automatically off-loaded. Function Off-loader generated a wrapper which reserved memory and transferred matrices to one GPU memory with the use of *cudaMalloc*

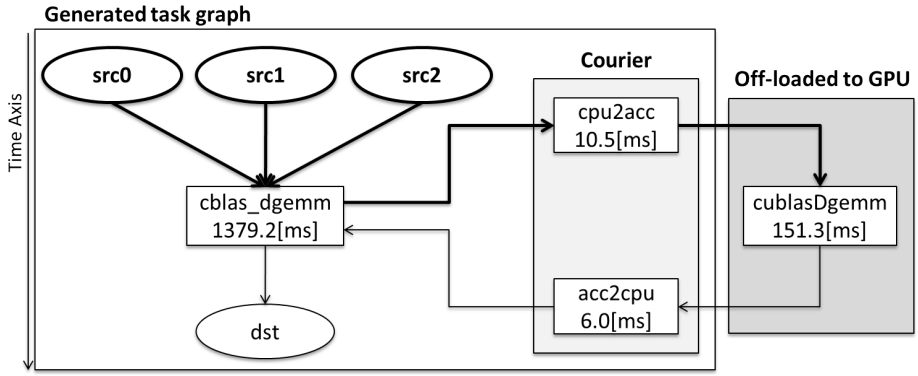


Figure 3.18: Generated task graph from dgemm binary.

and *cublasSet/GetMatrix*, respectively. Here no property change was required. For the IR description, Courier automatically introduced the *cpu2acc* and *acc2cpu* around the *cblas_dgemm*. In this case, all the acceleration processes were done by Courier, and there was no need for the user.

V. Results The right of Figure 3.18 shows the off-loaded flow. Processing time was shortened to 151.3[ms] by *cublasDgemm* on GPU. Including the data transfer and conversion time, total processing was sped up x8.16. Table 3.2 shows the 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 was 0.08[μs]. BLAS had much smaller overhead than that of OpenCV since it did not require data type conversion in this case. Consequently, overhead of Function Off-loader depends on the target function.

Table 3.2: Processing time comparison 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	168.7	168.7	166.9
Speed-up	x1.00	x8.16	x8.16	x8.26

3.4.3 Power Spectral Density Estimation (PSD) on a single mixed CPU-GPU platform

GNU Octave [55] 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 [56]. It included fast Fourier transform

(FFT) and other processing. FFT is a widely used routine for numerical analysis. Octave performs FFT by using fftw library [57] 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 [58] in CUDA 6.0 were used.

I. Analyzing running binary We prepared a tracing sub-program for FFT in Octave and it extracted the same type of information as for BLAS.

II. IR Description IR description was almost the same as BLAS case study. Octave included many other processes, but Courier could not analyze them since the current tracing sub-program doesn't support them. By adding information of other functions, applicability will be improved.

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

Listing 3.9: Detected processing flow in Courier IR.

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

IV. Acceleration with Courier Courier found that *fftw_execute_dft_r2c* was "safely off-loadable" and a corresponding accelerator function (fftw-compatible cuFFT) was available. Function Off-loader generated a wrapper includes the real-number input DFT function of cuFFT. IR description was also changed that 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 3.3 shows the final result. Note that "Total" is the whole 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 included pre/post-process. The overhead along with off-load was included in the result and *cpu2acc/acc2cpu* was zero. In this case, wrapper did not do any additional processing.

3.4.4 Multiple Sobel Operator on multiple mixed CPU-GPU platform

Evaluation environment is shown in Table 2.1. An equation 3.1 is used to measure speed up ratio.

Table 3.3: Processing time comparison 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

$$Speedup[\%] = \frac{T_{IAP}}{T_{DLP1} - T_{IAP}} \times 100 \tag{3.1}$$

An image filter called Sobel operators is used for the evaluation. It consists of three tasks, 50 times of x-axis Sobel operator ("xSobel"), 50 times of y-axis Sobel operator ("ySobel"), and generation of the output image in PNG format ("imout"). 100 images (1280x720 pixel) are processed. x-Sobel and y-Sobel run on GPU, and imout runs on CPU. Each task is implemented in OpenCV which is an open source library for image processing [59].

Six different versions of Sobel operator were implemented as described in Table 3.4. "CPU" running on only a CPU is the baseline. CPU is implemented in task level pipeline style, and used three physical threads. "DLP1" and "DLP2" are typical implementations for GPU, data level parallelism is exploited as shown in Figure 3.8. In the case of DLP1, on the CPU a thread to control one GPU runs. All 100 images are sent to the GPU, then x-Sobel and y-Sobel are applied sequentially. In the case of DLP2, two threads to control two GPUs run on the CPU. 50 images are sent to each GPU, then x-Sobel and y-Sobel are applied sequentially. Intel TBB's thread is only used to run threads. OpenMP or some other parallelise libraries are not used.

"IAP1", "IAP2" and "IAP2-P2" are implemented in IAP style. In these three versions, three tasks are applied to each input image in a pipelined manner. All tasks are registered to TBB and pipelined as I described in Section 3.3. Setting of TBB is decided to achieve the shortest execution time. Intermediate data of stages are stored in DDR3 on PEACH2 or DDR3 on CPU. IAP1 isn't fully implemented in IAP style since only one GPU is used and two tasks are switched in the single GPU. For IAP2 and IAP2-P2, all tasks run in a pipelined manner including data communication on GPUs. IAP2 which is shown in Figure 3.14, is assumed running on an ordinary computer node. It is necessary to copy inter-accelerator data to CPU to send another node. On the other hand, IAP2-P2 which uses PEACH2 is not required to copy the data to CPU. Additionally, the data are stored to DDR3 on PEACH2. Figure 3.15 depicts IAP-P2. Note that IAP1 sends back the intermediate date to CPU in order to make IAPs under fair condition.

Table 3.4: Six different implementation of Sobel operator

	Style	Used GPUs	Used memory
CPU	Task Pipeline	0	N/A (CPU)
DLP1	Data Parallel	1	N/A (CPU)
DLP2	Data Parallel	2	N/A (CPU)
IAP1	Task Pipeline	1	CPU
IAP2	Task Pipeline	2	CPU
IAP2-P2	Task Pipeline	2	PEACH2

From the viewpoint of data movement, DLP and IAP are completely different. The main drawback of DLP implementation is that, *imout* cannot start process until all the images are processed. On the other hand, IAP can start *imout* stage once after each image is finished to be processed in the former stages, since it is implemented in pipelined manner. Such sequential processing can be found in many applications.

Results Figure 3.19 shows a processing time and a chronological processing order of five GPU implementations. CPU only implementation takes 293.1s. It is not included in the figure. Implementations using GPU achieved at least 88.1% speed up compared to CPU. Processing time to apply three processes to each image is as follows, xSobel is 0.12 s, ySobel is 0.12 s, and *imout* is 0.04s. These numbers are the same in all implementations. The difference is processing order and the number of used GPUs. Note that, data transfer time is not shown in Figure 3.19 due to space limitations.

Processing time of this application is short, namely, data transfer time accounts for larger portion of total processing time. Longer data transfer time and short processing time usually cause negative effect to the total processing time, however IAPs show speed up. That is, I can expect that applications which have the opposite feature will show better result. In the case of data level parallel implementations, processing time of x/y-Sobel of DLP2 takes only half of that of DLP1, since evaluation application has no task and data dependencies. By contrast, the whole processing time of DLP2 doesn't take half of that of DLP1 since *imout* cannot start processing until two Sobel operators are finished. In the case of inter-accelerator pipeline, IAP1 achieved 33% speed up compared to DLP2. IAP1 which is implemented in pipelined manner successfully hides the data transfer time and the processing time of *imout*. The scheduler inside a GPU switches two Sobel operator tasks. IAP2 and IAP2-P2 achieved 52% speed up compared to DLP2, and also achieved 28 % speed up compared to IAP1. These implementations do not require task switch. Specifically, advantage of task level pipelining on multiple accelerator is proved.

Difference between IAP2 and IAP2-P2 is the place to store 2MByte inter accelerator data. In the case of IAP2, round trip data transfer between GPU and CPU occurs twice as shown in Figure 3.14. It takes approximately 5,000 μsec in total. Table 3.5 shows more specific time of data transfer to send 2MB to each DDR3.

Data transfer time of IAP2-P2 is a bit slow compared to IAP2. There are some reasons. First of

Table 3.5: Transfer time of 2MB data to each memory (inside a node)

	CPU to GPU	GPU to CPU
Time [μsec]	1,170	1,340
	GPU to PEACH2	PEACH2 to GPU
Time [μsec]	2,200	1,600

all, PEACH2 uses PCIe Gen2 x8 and CPU-GPU uses PCIe Gen3 x8. Furthermore, another reason is the small size of Intel Sandy Bridge's PCIe buffer. In addition, currently our data sending/receiving API for PEACH2 is beta version. Burst transfer is not fully supported. More precisely large data is divided into 4 byte data and transferred. Note that, this number is inside the single node, data transfer time of CPU and GPU to another TCA nodes becomes larger. In contrast, transfer time of GPU and PEACH2 is constant even if GPU sends the data to other TCA nodes.

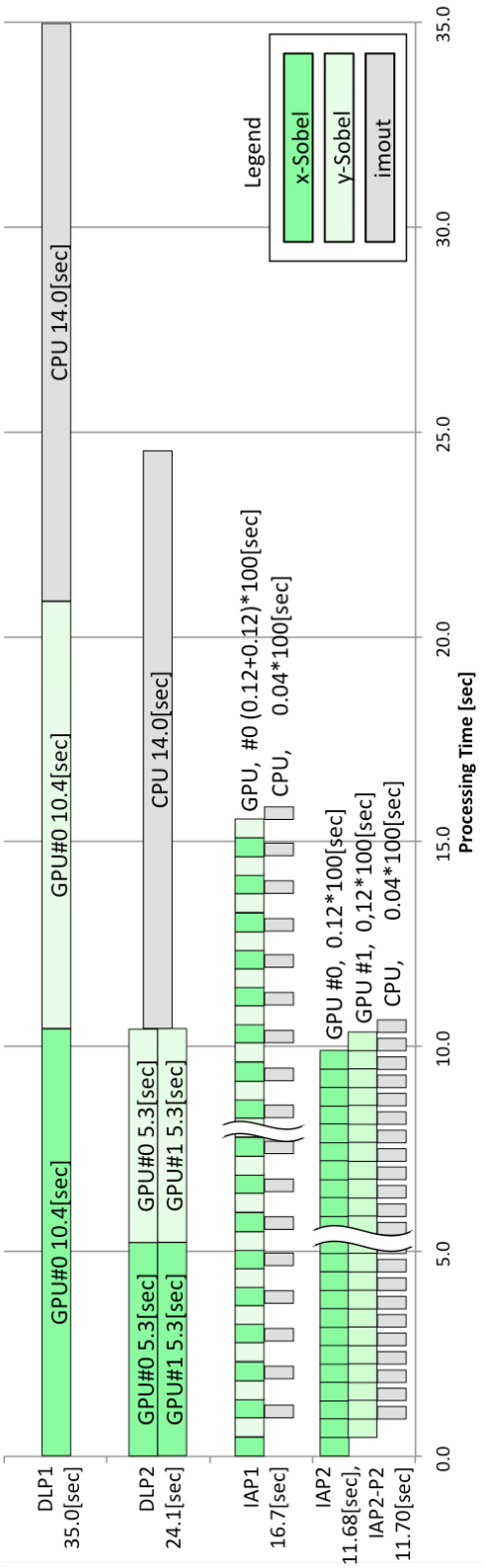


Figure 3.19: Processing time comparison and chronological processing order. IAPs are faster than DLPs at least 33%. IAP2-P2 which stores data to PEACH2 is almost the same as IAP2 which stores data to CPU. DLP1/2 cannot start *imout* until x/y-Sobel are finished. IAP1/2 and IAP2-P2, pipelined implementations, can start *imout* once after each image is applied x/ySobel. Processing time of each task for one image is as follows, xSobel is 0.12 s, ySobel is 0.12 s, and imout is 0.04s.

3.5 Related Work

3.5.1 Toolchains for supporting off-loading

There is a significant amount of existing researches on automatic off-loading systems [12] [13] [8] [15] [16] [40] [41] [60] [61].

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* [12]. For automatic distribution, it first requires a training run to build a database of relationships between the size of a data set 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 3.1.3. This did not include a system to determine the off-load parts as well as [13] or extract processing flow.

For a mixed CPU-FPGA platform, *DARES* [14] by Andrew Milakovich et al., *Hthread* [8] by Andrews et al., and *FUSE* [15] by Aws et al. are typical examples. DARES is one of a state-of-the-art software and hardware co-design 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 conventional 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 I know, there is no other similar work that can accelerate running binaries without accessing the original source code. Most common off-loading systems did not touch importance to function call graph with data and its data transfer time.

3.5.2 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 [11]. 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 [62] [63]. Purpose of them are to fully replaces an original function. Moreover, data transfer and processing flow are not a matter.

Backend uses DLL hijacking as a basis for realizing dynamic off-loading, but it didn't directly use it. One of the important problems of today's application acceleration is data transfer time. Once DLL hijacking fully replace an original function, data transfer time easily degrades performance. Furthermore, Courier has to maintain original processing flow after off-loading. Thus, I proposed Off-load Switcher to address these problems. In addition, DLL hijacking technique can be widely used in Linux. This means that Courier potentially supports many platforms. Heterogeneous platform which has CPU that runs Linux and accelerator emerged even in embedded area.

3.5.3 Related techniques used in Task Pipelining

Little research has been done on this topic for multiple GPUs. AXEL [22], the heterogeneous supercomputer cluster from Imperial College, has Map-Reduce framework to run application efficiently over multiple AXEL nodes. They implemented N-body simulation while exploiting data parallelism by using the framework. Although data parallel application works well in the framework and AXEL nodes, stream application is hard to apply this environment. Though AXEL has an inter-node communication network (PCI-X 2.0, 1.066Gbps at maximum [64]), they didn't use it in the paper. Huynh et.al. proposed a framework for implementing stream application on multiple GPUs [65]. They focused on programming model to exploit task parallelism and to describe stream application.

In the area of high performance computing with multiple FPGAs, Sano et al. proposed a similar concept in multiple FPGAs environment [66]. They focused on time axis task parallelism, and achieved strong scaling on stencil computation of CFD while keeping memory bandwidth constant. Their benchmark application is relatively simple and the target platform is FPGAs. That is, problems on communication between accelerators, inter-accelerator data, and pipelining become simple. In their case, FPGAs are connected via two 1GB/s uni-directional links (High-speed Terasic Connectors). Connection doesn't become bottleneck of this system since inter-accelerator data is divided into small pieces to fully utilize FPGA benefit. Storage which stores inter accelerator data is consisting of registers, thus the latency is very low. TCA can deal with more various applications since it has CPU, GPU and FPGA. However, the above three problems (communication between accelerators, inter-accelerator data, and pipelining) are difficult to solve.

3.6 Chapter Summary

This chapter 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 consist of three main 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. We also proposed a task level pipelining on multiple GPUs in TCA node. Inter Accelerator Pipelining (IAP) is an implementation concept that connects multiple accelerators as a form of a pipeline computation. We implemented a simple image processing application on a single TCA node in IAP style to evaluate the concept. We also consider three problems associated with the pipeline implementation. The problems are communication between accelerators, storage which stores inter accelerator data, and pipelining itself. We solved them by using PEACH2 and Intel TBB. Finally, three application binaries of HOG, dgemm and PSD are accelerated by using Courier on CPU-GPU environment. Simple image processing application is also accelerated by using IAP implementation not data parallel implementation.

Chapter 4

Courier-FPGA: A Toolchain for Mixed Software Hardware Pipeline on a CPU-FPGA Platform

This chapter proposes yet another new tool chain, called *Courier-FPGA*. Courier-FPGA is based on Courier and its target heterogeneous platform is a mixed CPU-FPGA. It 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.

Courier-FPGA is first presented. It includes special features that treat running binaries and accelerate them by replacing software functions with the built pipeline including pre-defined hardware modules. Then, this chapter describes the details of mixed software hardware pipelines on CPU-FPGA platforms in Section 4.2 and Section 4.3. Section 4.4 gives a case study showing the capability of Courier-FPGA.

A short summary of this chapter and the differences between Courier-FPGA and the original Courier are as follows:

- The 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.

4.1 Courier-FPGA: A Toolchain for Mixed Software Hardware Pipeline

Courier-FPGA is based on Courier, a toolchain for a single accelerator like GPU. This section describes the detail of Courier-FPGA and its features to make the best use of CPU-FPGA platforms. Please refer back to Chapter 3 for details of the original Courier.

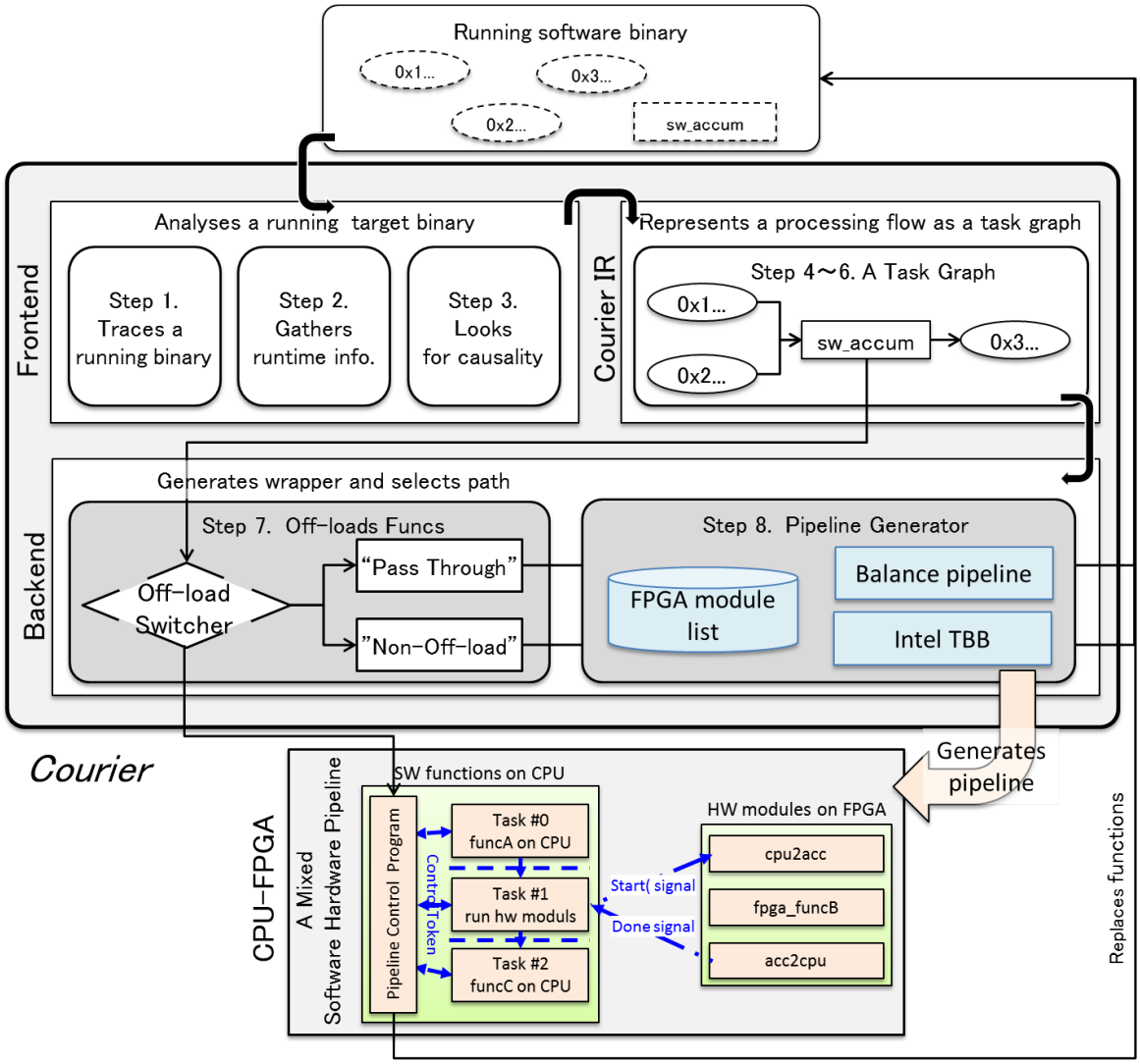


Figure 4.1: Overview and work-flow of Courier-FPGA: the *Frontend* analyzes the running binary (Step1, 2 and 3) and then generates a task graph and a *Courier Intermediate Representation (IR)*. The user refers to the graph and results, decides which parts to off-load and rewrites IR if needed (Steps 4, 5, 6 and 7). After that, the *Pipeline Generator* builds a mixed software hardware pipeline (Step 8). Finally, the *Function Off-loader* replaces function and off-loads it to the accelerator (Step 9).

4.1.1 Overview of Courier-FPGA

A motivation of the "original" version of Courier was to provide a simplified work-flow of application acceleration for non-expert users. The original Courier requires a target binary and pre-defined

corresponding functions of the accelerator. A user only designates a running target binary to the original Courier. The original 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. The 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 *Function Off-loader*, *Courier-FPGA* can build a mixed software hardware pipeline on a CPU-FPGA platform.

Figure 4.1 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 task graph 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 their properties in the graph during the acceleration process to decrease the number of data transfers between the CPU and accelerator.

- *Courier Intermediate Representation (IR)* is a simplified language that enables users to modify processing flow and designate parts to off-load to the Backend if needed.

- The *Backend* makes a mixed software hardware pipeline. It automatically off-loads the function, if the corresponding function is ready for the accelerator. The *Function Off-loader* automatically decreases the number of data transfers along with off-load, and maintains the original processing flow before and after off-load. The original Courier can deal with GPU, while *Courier-FPGA* can deal with FPGA.

The Backend can be divided into two steps that corresponds to Steps 8 and 9 in Figure 4.1. In Step 8, the *Pipeline Generator* builds a mixed software hardware pipeline. It first generates the corresponding hardware module on an FPGA, and then prepares software functions and a pipeline control program. Then, the *Function Off-loader* selects a path and replaces functions with the generated pipeline in Step 9. These two functions in the Backend, the *Pipeline Generator* and *Function Off-loader*, are originally developed for Courier-FPGA.

4.2 A Mixed Software Hardware Pipeline

4.2.1 Fundamental Concept

The Backend automatically builds a mixed software hardware pipeline and off-loads the functions to the pipeline after the Frontend analyzes the running binary and makes a task graph. 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 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 [45]". Unlike a single GPU, the off-loading target of the 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 4.2 shows an example of automatic off-loading by using the *Pipeline Generator* and *Function Off-loader*. The Structure of a built pipeline; a mixed software hardware 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 software hardware pipeline. First, the Pipeline Generator automatically generates a code of pre-defined corresponding hardware modules, configures them on the FPGA, and prepares 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 at the bottom of Figure 4.2, "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 4.3 shows a typical case of building a mixed software hardware pipeline.

The following sections describes how the Pipeline Generator automatically builds an efficient mixed software hardware task pipeline, and how the Function Off-loader performs off-loading automatically.

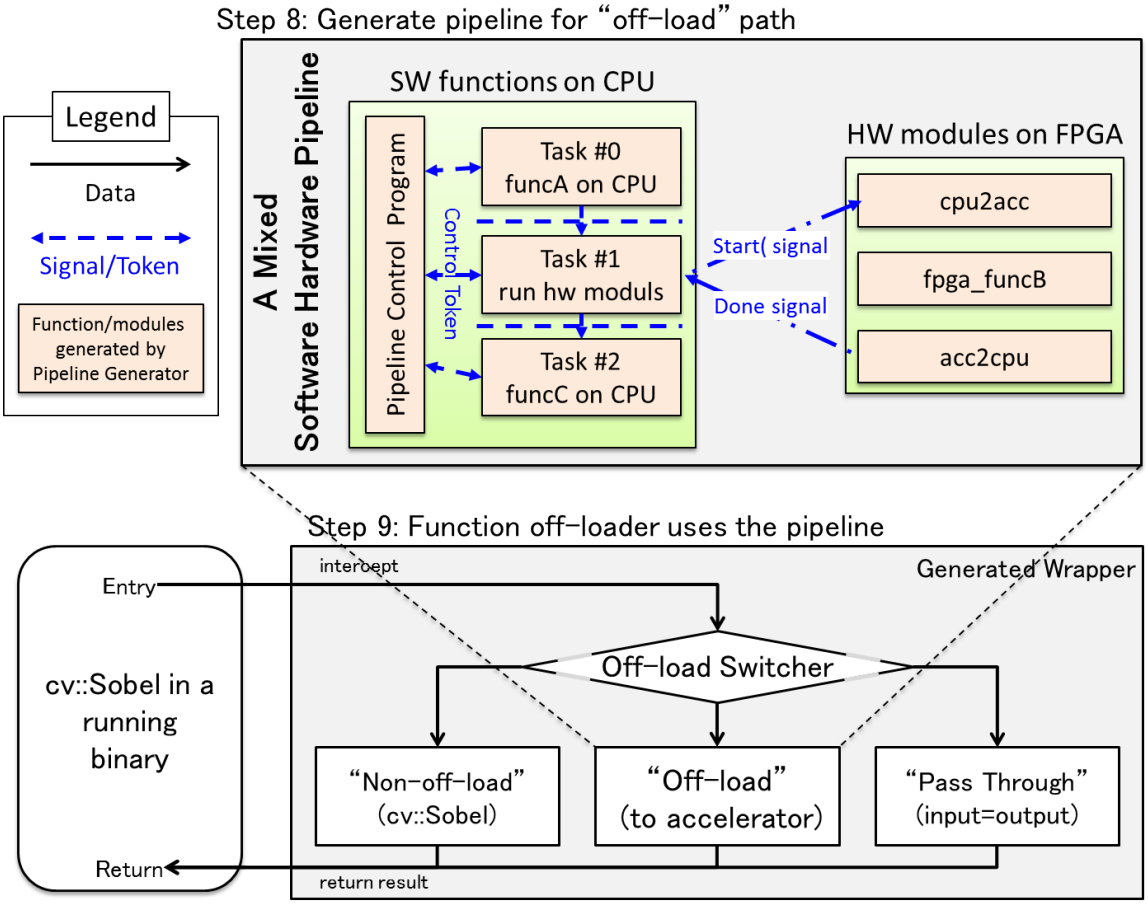


Figure 4.2: In step 8, the Pipeline Generator prepares a mixed software hardware task pipeline. In step 9, the Function Off-loader uses the pipeline when "off-load is selected".

4.2.2 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 [3] and Arria V SoC [4] 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 [46], Boost::thread [47], OpenMP [48], or Glib::thread [49]. However, they are not intended for pipelined execution of tasks. OpenMP requires tricky programming or unofficial extension to realize pipeline execution [50] [51], since it is intended to exploit loop level parallelism. The others are developed for exploiting thread level parallelism, and not suitable for task level pipeline.

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. TBB 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 4.3 shows the behavior of a typical mixed software hardware 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, I 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 D. Then, it generates source code for 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 a software function is made for it. 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 the first input data (data #0) from the running binary. Then, Task #0 first executes automatically 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 pipeline stall compared with the hardware pipeline. Additionally, stages which run in parallel can be automatically changed since a task is randomly assigned to an idle thread by the control program.

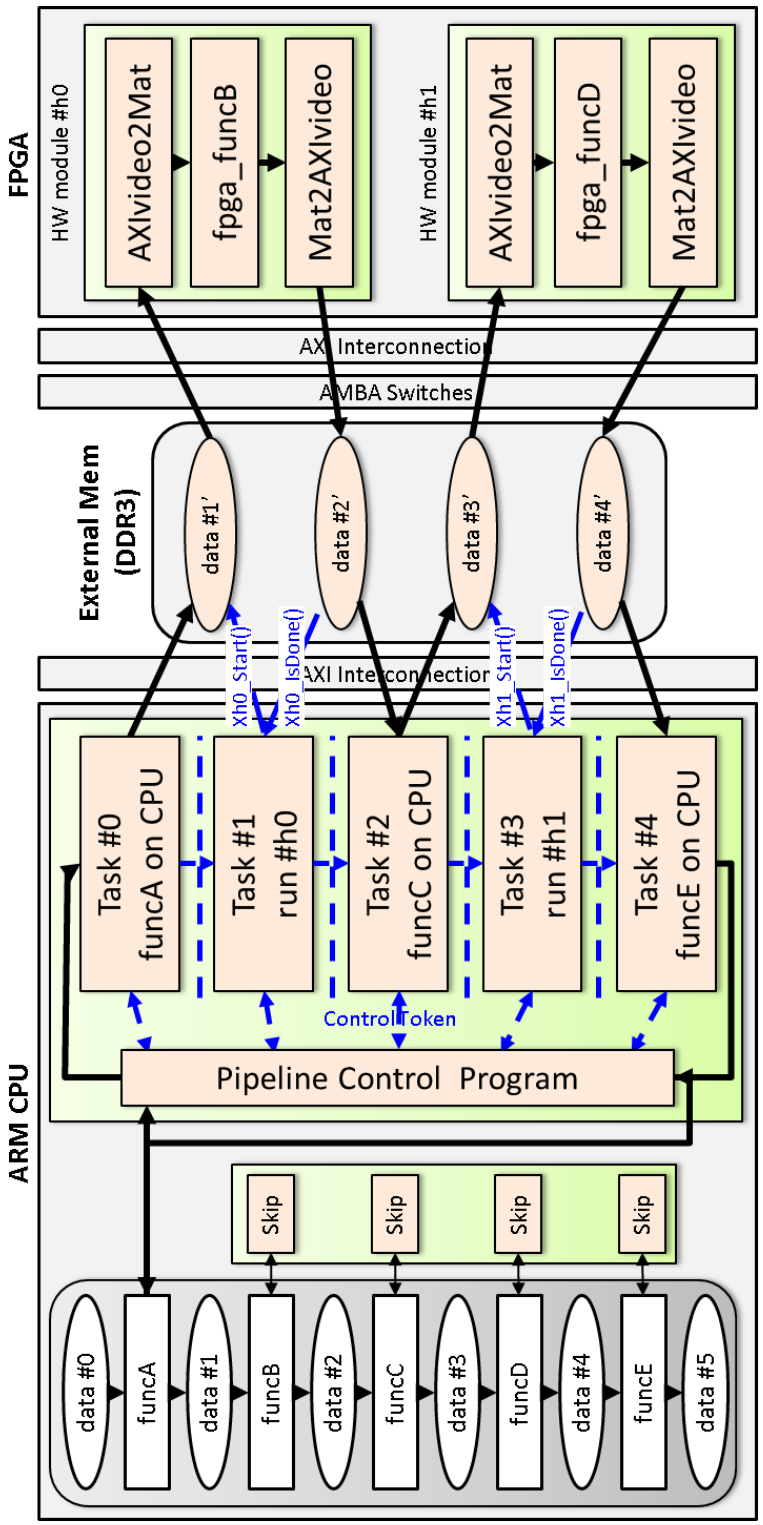


Figure 4.3: Behavior of a typical mixed software hardware pipeline controlled by software program. Shaded rectangles are generated by Pipeline Generator. Tasks run in a pipelined manner, and each task can send and receive input and output data which is indicated by bold line. Input and output data of 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.

4.2.3 Building an Efficient Mixed Software Hardware Pipeline

When we build a mixed software hardware pipeline, we have to consider concurrency and the number of threads in order to make it efficient. Considering these problems is equal to a decision of which tasks can run in parallel and how to divide the extracted flow into some stages. The following solutions are proposed and implemented them in the Pipeline Generator so as to automatically generate an efficient pipeline.

4.2.3.1 Concurrency

A feature of the mixed software hardware pipeline is that stages which run in parallel can be automatically 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 Intel 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.

4.2.3.2 Number of Threads

The number of tasks which can run in parallel, depends on the number of logical CPU threads on the platform. The number must be defined to make the task pipeline with Intel 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.

$$T_{stage} = T_{total} \div (N_{logical\_thread} + 1) \tag{4.1}$$

where T_{stage} is the target time of each stage, T_{total} is the 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.

4.2.4 Generating Code for Hardware Module

For each hardware task in Section 4.4, it is used that an OpenCV-compatible high-level synthesis library provided by Xilinx [6]. 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 [67] 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 software hardware 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 require 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.

4.2.5 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. Such a mechanism can be applied to any binary without re-compilation and is supported in most Linux environments.

An example wrapper is shown at the bottom of Figure 4.2. 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 and the role of each path is the same as Section 3.2.

4.3 An Automatic Mixed Software Hardware Pipeline Builder

In Step 8, the *Pipeline Generator* automatically searches corresponding predefined hardware modules from a database by function name, places them on FPGA and prepares software functions. Then, it makes a software program that runs mixed software hardware tasks in parallel so as to make the best use of the parallelism. The parallel tasks perform processing which corresponds to a target binary. In Step 9, *Function Off-loader* wraps the built pipeline and actually replaces the function in the target binary with the pipeline. In this step, off-load is ready to deploy. During deployed run, the control program made by the Pipeline Generator starts to accelerate the binary. Even if the functions in the target binary run sequentially, Courier-FPGA can run them in a pipelined manner.

4.3.1 Structure of a mixed software hardware pipeline

Structure of a mixed software hardware pipeline on a CPU-FPGA platform is composed of the following three main parts. The control program is needed in order to run multiple software and hardware functions in a pipelined manner. Software functions are used when corresponding hardware modules do not exist in a database.

- A pipeline control software program that controls the software hardware tasks.
- Software functions run on the CPU.
- Hardware modules run on the FPGA.

In Figure 4.3 the Pipeline Generator generates five-stage pipeline, two hardware modules and three software functions. The pipeline control program runs these tasks in a pipelined manner. On a deployed run, tasks work as follows from the viewpoint of the target binary. The Function Off-loader hooks funcA and its input data (data #0) from the running functions in the target binary, which is illustrated on the left of the figure. Then, Task #0 processes funcA and stores the result (data #1') in an 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 and stores a result data (data #2') in the memory. 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.

4.3.2 Building a mixed software hardware pipeline

Figure 4.4 shows a processing flow of how the Courier-FPGA automatically builds a pipeline. First of all, the Frontend analyzes a target binary and makes a list of running functions name. Then, the Backend searches corresponding modules from a hardware module database. Found corresponding modules are synthesized and placed on FPGA. On the other hand, functions which cannot be found

become software ones running on CPU. Finally, Pipeline Generator builds a balanced pipeline considering processing time of functions/modules. Each stage of the pipeline runs hardware module(s) on FPGA or software function(s) on CPU.

In the case of Figure 4.3, the Backend found two corresponding hardware functions: funcB and E in the database. Then, Pipeline Generator generates two hardware modules: the former contains fpga_funcB, and the latter contains fpga_funcD. In addition, Task #1 and Task #3, a software part of these modules, were also prepared. Software parts perform communication between hardware modules and an external memory. On the other hand, there were no hardware modules for funcA, C and E in the database, so software functions are used for them. Thus, five tasks, two hardware modules and three software functions, are used for the five-stage pipeline. The pipeline control program runs these tasks in a pipelined manner and keeps the order of stages.

4.3.2.1 Generating hardware modules

As I described above, the Backend searches a corresponding hardware module in a database for each analyzed function. Technically, this database includes an OpenCV-compatible high-level synthesis library provided by Xilinx [6]. The Pipeline Generator generates code for the hardware modules, input/output port and optimization pragma. For the hardware modules, Pipeline Generator simply generates the corresponding module name. For example, *hls::Sobel* is used for *cv::Sobel* as a corresponding function and arguments are defined. For the input/output port, Pipeline Generator uses *AXIvideo2Mat* and *Mat2AXIvideo*. These are the code of AXI4-Streaming protocol including Video DMA controller that communicates the CPU and the hardware module. For the optimization pragma, Pipeline Generator inserts *#pragma HLS dataflow* by default in order to achieve shorter processing time.

In the case of Xilinx's OpenCV library, each function is optimized for per pixel processing. In addition, 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. The Pipeline Generator automatically calculates and defines the width 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.

Generated hardware modules are prepared as a block device on Linux, and a basic device driver APIs that send/receive data are prepared by Xilinx's high-level synthesis tool. In the case study, *XTask0_Start()* function sends input data to start process on 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 CPU side.

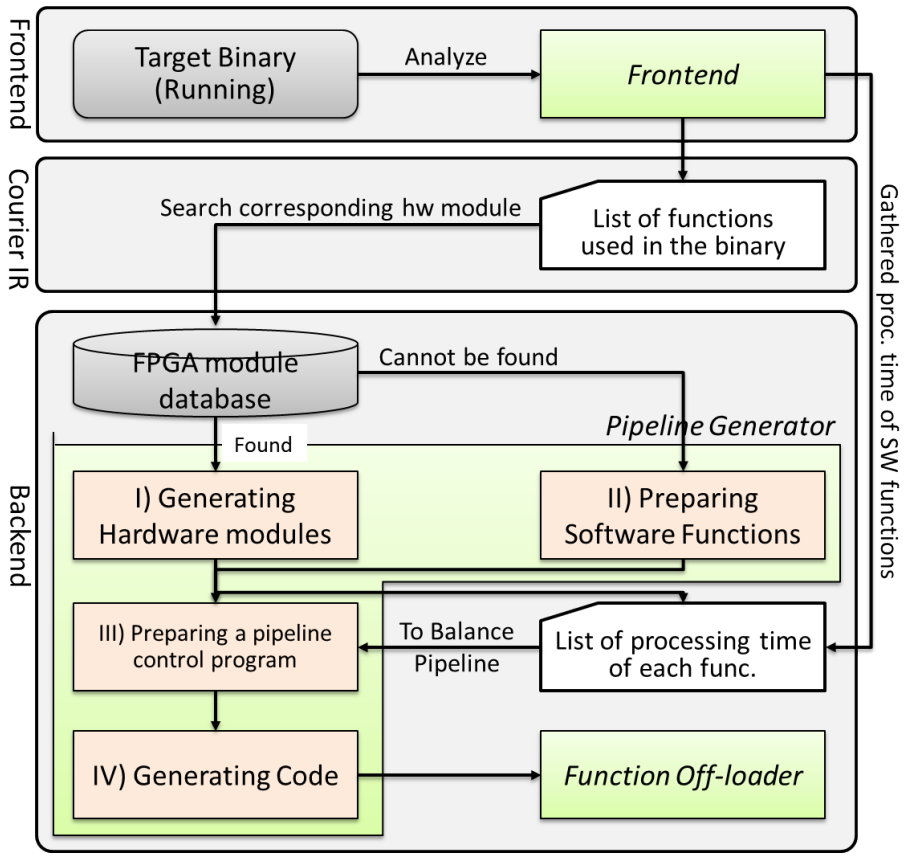


Figure 4.4: A processing flow of building a mixed SW/HW pipeline. Shadowed rectangles are parts of the Courier-FPGA and ones filled with oblique lines are explained in Sect.4.2 B.

4.3.2.2 Preparing Software functions

For each software function task, the Backend prepares to run original functions in the binary and dynamically replaces them during deployed run. The Backend first looks for a function name in function libraries by using *dlsym* [68] with "RTLD_NEXT" option so as to use an original function. The function library is designated by *dlopen* with "RTLD_LAZY" option [68]. The reason why Courier used such method is that Function Off-loader basically uses a software technique called *DLL injection* so as to realize dynamic off-loading. When we generically use DLL injection and want to use software functions in the target binary, the above described method is required. In the case of hardware modules, we don't need to use this method.

4.3.2.3 Preparing a pipeline control software

For a pipeline control software program, it is used that Intel Thread Building Blocks (TBB) that runs multiple functions in a pipelined manner on CPU. The *tbb::pipeline* class is provided to build a pipeline. A user can add an arbitrary task to each stage of the pipeline skeleton, and specify the processing order 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 and the users write codes for each slave thread (e.g. software functions or hardware modules). TBB is also capable of double buffering when two or more tasks are running.

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 pipeline stall compared with the hardware pipeline. Additionally, stages which run in parallel can be dynamically changed since an idle thread is randomly chosen by the control program.

4.3.2.4 Generating Code

I use Python and Jinja2 [69] to implement the Pipeline Generator. The Pipeline Generator is a script and technically composed of a pipeline skeleton part and a task part for each stage. The former is almost static and the latter is flexible since it contains above described software functions and hardware modules. The Backend tells information (e.g. a filter type of Intel TBB, data type of input/output or actual processing code) to the Pipeline Generator, and it generates the whole code.

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 thread plus one and searches the closest sub-total of processing time of functions". The policy is derived from the following considerations. According to our preliminary evaluation, the number of stages should close to that of a logical thread of the Zynq (= 2). This is because the control load of master thread and the data transfer frequency of intermediate data should be reduced for a streaming architecture. 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. Additionally, the Pipeline Generator defines the first and last functions to run serially run (*serial_in_order*) , while the rest of the functions run in parallel (*parallel*).

4.4 Case Study

This section illustrates our work-flow by describing a practical case study. Three case studies, a histogram of oriented gradients (HOG), a Harris Corner Detector, a 3D Object Rotation, are conducted. 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 667MHz with 512MB memory (called PS: Processing System) and 85,000 Series-7 programmable logic cells (called PL: Programmable Logic). Linaro 32bit (Debian 7.0) ran on the PS. Synthesis tools are Xilinx Vivado HLS and Vivado 2014.2. OpenCV version 2.4.8 is used.

4.4.1 Histogram of Oriented Gradients (HOG in OpenCV)

4.4.1.1 Acceleration Work-flow of Courier-FPGA

I. Analyze running binary This step is the same as the HOG case study in Section 3.4.1. Although, the target binary was running on ARM CPU, the same tracing subprogram for OpenCV can be used in this case.

II. Generating the task graph of the running binary After the profile run, a task graph of the running binary was automatically generated (see the left of Figure 4.5). The user examined the graph and decided whether to off-load and non-off-load parts if needed. The graph was identical to the previously described processing flow in Section 3.4.1. 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 $\times$ width $\times$ bit-depth $\times$ channels; e.g., the first node was $1280 \times 720 \times 32\text{bit} \times 1$-channel). The processing time is shown in the second row of the ellipse node. Nodes were aligned in chronological order. According to the graph, each input image was 1280×720 and processed in 650,856 [μs] in total. This was less than 1.5 frames per second ([fps]). The Courier IR description was automatically generated. Users can modify this to change the actual processing flow if needed. The details of the IR in the case study was the same as Section 3.4.1.

III. Acceleration In this step (Steps 8 and 9 in Figure 3.1), Courier-FPGA first searched for "safely off-loadable" parts, where a processing flow was straight-forward, functions and input/output data are both traced, and a corresponding accelerated hardware module was available. For such parts, Courier-FPGA automatically built a mixed sw/hw pipeline by using the Pipeline Generator and used it by using the Function Off-loader in default mode.

In this case, the Pipeline Generator generated a four-stage mixed software hardware pipeline. Each processing step was assigned to a task of the pipeline. Tasks #0 and #2 could be off-loaded to the FPGA since the corresponding hardware modules were available. But inside functions of both tasks could not be pipelined by using *#pragma HLS PIPELINE* because of branching and converging.

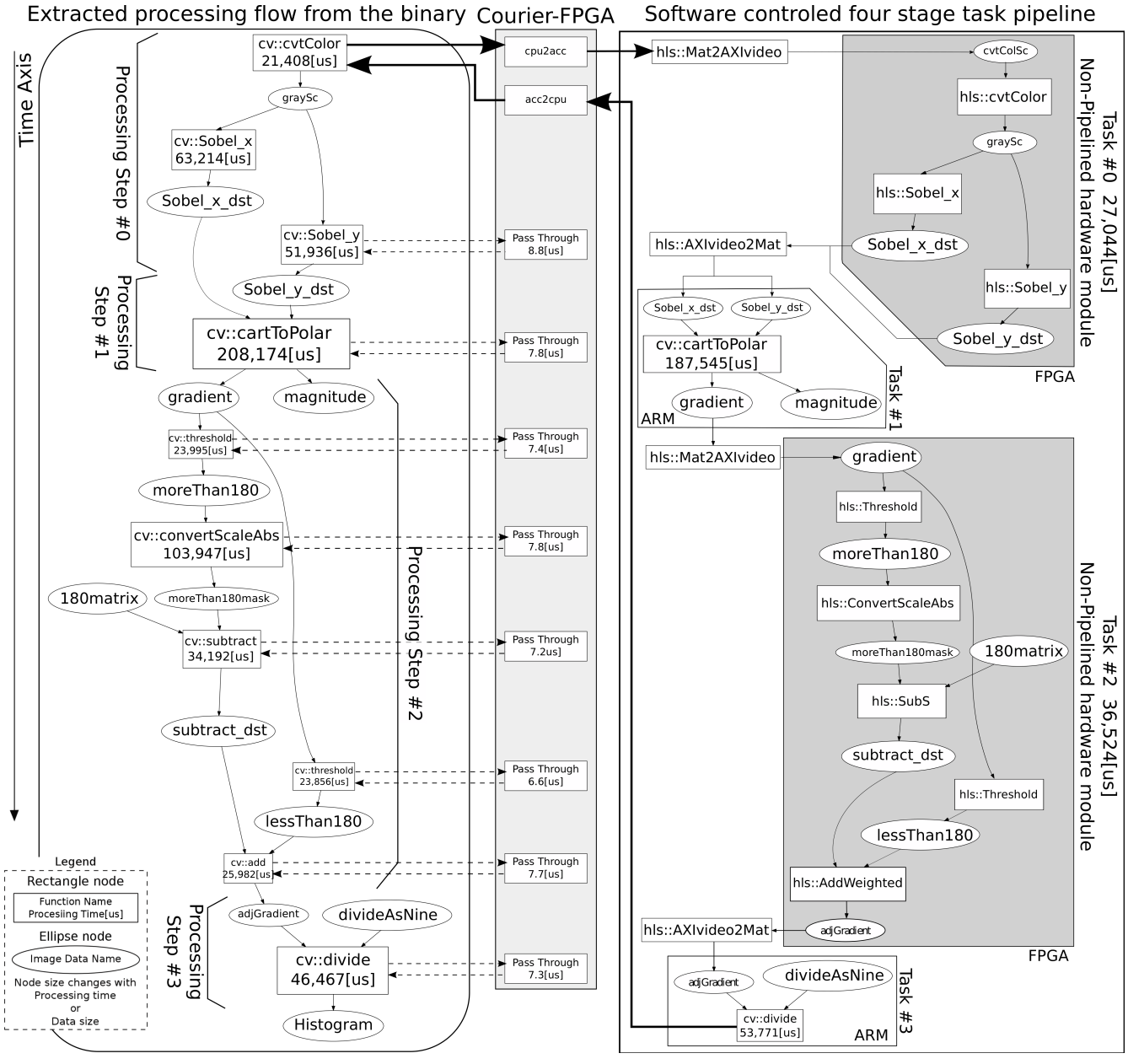


Figure 4.5: Processing flow extracted from the running binary (left) and off-loaded flow (right). Each processing step is assigned to a task. The Function Off-loader generates a four stage mixed software hardware pipeline.

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 has two logical threads. The Function Off-loader intercepted *cv::cvtColor* as "the head" of a series of functions and off-loads it. For the rest of the functions, Courier-FPGA intercepted and passed them on to maintain the original processing flow by selecting "Passes Through". If the functions were not passed, they were off-loaded and used in the original binary.

IV. Results The right side of Figure 4.5 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 was the same as the original one, predefined accelerated modules were run on a PL of Zynq.

Table 4.1: Processing time comparison ([μs])

	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

Table 4.1 shows the average processing times when of 200 video frames. Courier-FPGA shortened the processing time to 163,510[μs] and achieved a 6.1[fps], or x3.98 speedup compared with the original binary. In Table 4.1, "Original Binary" indicates the target binary running on the CPU, and "Courier-FPGA" is the final result. AXIvideo2Mat is input to the hardware module via AXI bus and Mat2AXIvideo.

The generated four-stage mixed software hardware pipeline worked well. "Total (Average)" is smaller than the pipeline's Task #1 since TBB searched for and run an idle task from the thread pool. According to our processing log, Task #0 run multiple times and stores multiple results while Task #1 run. 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 became shorter than the time taken by Task #1. Figure 4.6 is a graph showing the relationship between processing time per frame and the number of processed frames. The graph shows 155,000[μs] is the lower limit for this task pipeline.

Tables 4.2 and 4.3 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 transfers via the AXI Stream bus). In the case of Task #2, there was no Mat2AXIvideo in Table 4.3 because

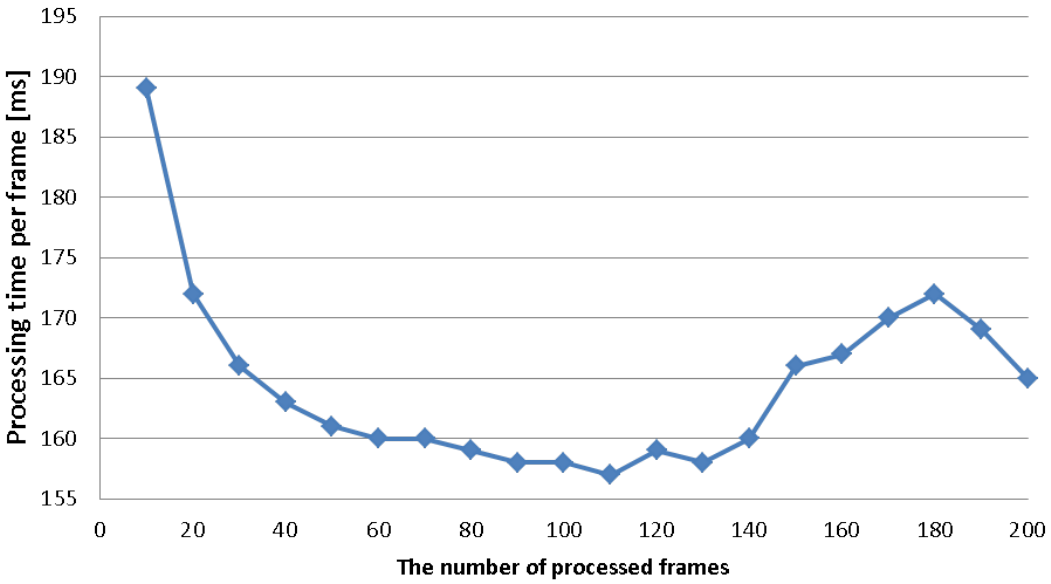


Figure 4.6: Processing time per frame fluctuated a little since Intel TBB's pipeline is based on a thread pool. It works differently from a pure hardware pipeline.

Table 4.2: Evaluation: Frequency, Latency and Exec. time

Module	Freq. [MHz]	Latency [clk]	Exec. time [μs]
Task#0	172.1	4,654,817	27,044
Task#2	152.4	5,567,778	36,524

Table 4.3: Evaluation: 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
convertScaleAbs	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

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 were 8 bits. On the other hand, those of Task #0 were 32 bits and 8 bits.

4.4.2 Harris Corner Detector (cornerHarris in OpenCV)

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 4.4. Inputted image size was 1920 x 1080. Courier-FPGA built a three-stage pipeline, and x22.1 speed-up was achieved compared with the original binary.

Table 4.4: Processing time comparison ([μs])

	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	—

4.4.3 Rotate 3D Object (glRotatef in OpenGL)

hello_world_in_glsl is a simple program of OpenGL and can be downloaded from the website [70]. Four functions listed in Table 4.5 are targeted. We implemented a corresponding hardware module of *glRotatef* that performs some single precision floating point matrix calculation [71]. Courier-FPGA built a single-stage pipeline because of the data structure of OpenGL. A 1.29 times speedup was achieved.

Table 4.5: Processing time comparison ([μs])

	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	—

4.5 Chapter Summary

This chapter presented *Courier-FPGA*: a new toolchain for mixed software hardware pipeline 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 generated 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.

Chapter 5

Conclusion

In this chapter, it is first summarize and discuss our three proposals in Section5.1:*Courier*, *Courier-FPGA*, and *Inter Accelerator Pipeline*. Advantages and disadvantages are given as well. Then, the thesis is concluded in Section 5.2.

5.1 Summary and Discussion

5.1.1 Toolchain for Automatic Function Off-load on a CPU-GPU Platform

Chapter 3 described detail features of Courier. Courier is composed of *Frontend (Runtime Analyzer)* , *Courier IR* and *Backend (Function Off-loader)*. Frontend automatically traces a target running binary and tries to detect a processing flow of functions by using a heuristic approach. Courier IR represents the processing flow in a graph and code. A task graph is constructed to understand the flow and find parts that should be off-loaded. Backend automatically off-loads the functions by using the Function Off-loader. It automatically deals with problems during off-load such as the number of data transfers or unconditional off-loads. Courier successfully shortens the processing time of three computational intensive applications (HOG, BLAS and FFT) on a single mixed CPU-GPU platform. For a multiple mixed CPU-GPU platform, it is proposed that a task level pipelining technique that enhances the performance of stream computation applications.

The applicability of Courier is detailed in Section 3.4. The current version of Courier can accelerate well-known functions and has a straight-forward processing flow. Additionally, Courier can support an arbitrary library by introducing a new “add-on”. On the other hand, there are some limitations as this thesis discussed in Section 3.1.6. The biggest limitation is that corresponding accelerator functions must exist beforehand. Courier-FPGA can shorten the processing time without the corresponding functions by building a pipeline on a mixed CPU-FPGA platform. This mechanism needs to be researched for this platform. Other problems are SAME-INOUT limitation and flexibility limitation of functions within control statements. The SAME-INOUT limitation is that the number of input/output of replaced functions must be the same as that of original functions in the binary. To

solve this limitation, a certain kind of data management mechanism is required. The flexibility limitation of functions within control is that Courier cannot recognize the control statements. Thus, the efficiency of automatic off-load is degraded. In addition, some inefficiencies exist on Courier as well. If the target functions have many inputs/outputs or include a complicated processing flow, Courier can technically handle them but will not be efficient. To deal with these problems, Courier needs to be enhanced the features of Frontend and Backend.

5.1.2 Task Level Pipelining on a multiple CPU-GPU Platform

Chapter 3 proposed an *inter accelerator pipelining (IAP)* on a multiple mixed CPU-GPU platform. IAP is a special case of task level pipelining and all tasks within inter/intra-node run in a pipelined manner. Each stage of IAP corresponds with assigned tasks on each GPU. IAP is suitable for stream computing applications such as computational fluid dynamic or image processing. Note that IAP does not replace data level parallel implementation but complements it. Although IAP can be applied to an ordinary multiple CPU-GPU platform, data transfer among nodes degrades the total performance. The TCA cluster, that enables the ultra low latency direct communication among multiple GPUs within the cluster from the University of Tsukuba, is adopted. On TCA, this thesis shows that the processing time of simple image filtering can be made shorter than that of an ordinary CPU-GPU platform.

IAP is applicable in many stream computation applications, but the applicability of IAP must be studied in more detail. For example, the break-even point of communication latency and processing time of each task to automatically divide the stage.

5.1.3 Toolchain for Mixed Software Hardware Pipeline on a CPU-FPGA Platform

Chapter 4 explained features of Courier-FPGA in detail. The target platform of Courier-FPGA is a mixed CPU-FPGA platform. The main contribution of Courier-FPGA is building a mixed software hardware pipeline on the platform by using a *Pipeline Generator* and Function Off-loader. Courier-FPGA shares Frontend and Courier IR with original Courier. Analyzed processing flow is divided into some tasks, and each task is assigned to each stage of the pipeline. Even if the corresponding hardware modules do not exist, Courier-FPGA can shorten the processing time by pipelining the processing flow. The pipeline is controlled by a master thread and can have software functions on a CPU and hardware modules on an FPGA. A key point of the Pipeline Generator is balancing of each stage. By using the actual information of processing time collected by Frontend, the Pipeline Generator builds a balanced pipeline and makes the best use of this platform. Practical case studies are conducted to confirm the applicability of Courier-FPGA in Section 4.4. Three applications are accelerated on a mixed CPU-FPGA platform.

A mixed software hardware pipeline successfully shortens the processing time, but there are some limitations. One is that the number of threads that can be run in parallel depends on the number of

logical threads of CPU. In the case of Zynq by Xilinx, it is two. I think that this limitation will be relaxed soon since we can obtain a ARM CPU that has four or more logical threads now. Another limitation is how to reduce the pressure on a CPU-FPGA bus. Embedded platforms often do not have powerful bus bandwidth. The bus usage needs to be maximized when the number of hardware modules increases.

5.2 Concluding Remarks

Heterogeneous platforms have become important in many area. In a scientific computation domain that requires a lot of processing power, a mixed CPU-GPU platform is going to make up an important share. In an embedded device domain that requires a power efficient system, a mixed CPU-FPGA platform has been developed. Many applications, not only compute-intensive ones but also data-intensive ones, can be accelerated on such platforms. However, although a mixed CPU-GPU platform and a mixed CPU-FPGA platform are different, they share the fundamental idea of off-loading time consuming parts onto accelerators and shortening the total processing time. Application acceleration on such platforms is a specialized task. Although many studies have been done to alleviate programmers burden, application acceleration is still difficult for non-expert users. Furthermore, if the programmers cannot access the original source code, acceleration is almost impossible. Against this background, the demand for simplifying the work-flow of application acceleration has increased.

This thesis proposed two toolchains for application acceleration: *Courier* and *Courier-FPGA*. Both toolchains are intended for non-expert users who do not have expertise in or special knowledge of target heterogeneous platforms. It is also proposed that an inter-accelerator pipeline (IAP) on a multiple mixed CPU-GPU platform. IAP forms a task level pipeline among multiple GPUs. Courier is designed to automatically analyze specific functions and data in a running binary and replace functions with corresponding accelerator functions if possible. The target platforms are a single mixed CPU-GPU platform and a multiple mixed CPU-GPU platforms. Courier-FPGA is based on Courier, but the target platform is a single mixed CPU-FPGA platform. It builds a function-level pipeline structure between the hardware modules on an FPGA and software functions on a CPU automatically. IAP provides yet another implementation methodology for stream computation applications on a multiple CPU-GPU platform. Each task assigned to each GPU among intra/inter-node works in a pipelined manner.

There are some future works to extend the applicability of proposed toolchains and implementation methodology. Dynamic program analysis performed by the Frontend of Courier needs to gather more information. If Courier can obtain more information of the target application, it can deal with more complicated processing flows such as loops or branches. I will research how to generate a more flexible task pipeline to relax user constraints by Backend, such as resource utilization or power consumption.

Bibliography

[1] TOP500 Lists | TOP500 Supercomputer Sites . http://www.top500.org/lists .

[2] Are Supercomputing's Elite Turning Backs on Accelerators? - HPC wire . www.hpcwire.com/2014/06/26/accelerators-hold/ .

[3] Zynq-7000 Programmable SoCs . http://www.xilinx.com .

[4] SoCs - Portfolio - Altera . https://www.altera.com/products/soc/portfolio.highResolution-Display.html .

[5] CUBLAS . https://developer.nvidia.com/cuBLAS .

[6] Xilinx Vivado Design Suite User Guide: High-Level Synthesis (UG902) . http://www.xilinx.com/support/documentation/sw_manua-ls/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf .

[7] Altera SDK for OpenCL - Overview . https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.highResolutionDisplay.html .

[8] D.Andrews, W.Peck et al . The Case for High Level Programming Models for Reconfigurable Computers . In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 21–32, 2006.

[9] Impulse Accelerated Technologies . http://www.impulseac-celerated.com .

[10] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364, 2010.

[11] Feng Li, A. Pop, and A. Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. In *IEEE Micro*, pages 19–31, 2012.

[12] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*, pages 45–55, 2009.

[13] M.Becchi, S.Cadambi and S.Chakradhar . Enabling Legacy Applications on Heterogeneous Platforms . In *The 2nd USENIX Workshop on Hot Topics in Parallelism (USENIX HotPar)*, 2010.

[14] Andrew Milakovich, Vijay Shankar Gopinath, R. Lysecky, and J. Sprinkle. Automated software generation and hardware coprocessor synthesis for data-adaptable reconfigurable systems. In *IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 15–23, 2012.

[15] I.Aws and L.Shannon, . FUSE : Front-end user framework for O/S abstraction of hardware accelerators . In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 170–177, 2011.

[16] R.Lyseckya, F.Vahida, S.Tan . A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA Compilation . In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–62, 2005.

[17] Brochure, Movie | Center for Computational Science, University of Tsukuba . http://www.ccs.tsukuba.ac.jp/eng/about-2/pamphlet-video/ .

[18] GeForce GTX TITAN Z Extreme Gaming Graphics Card | NVIDIA . http://www.nvidia.com/gtx-700-graphics-cards/gtx-titan-z/ .

[19] Programming Guide :: CUDA Toolkit Documentation . http://docs.nvidia.com/cuda/cuda-c-programming-guide .

[20] NVIDIA GPUDirect NVIDIA Developer Zone . https://developer.nvidia.com/gpudirect .

[21] Hanawa Toshihiro, Kodama Yuetsu, Boku Taisuke, and Sato Mitsuhisa. Interconnect for tightly coupled accelerators architecture. In *IEEE 21st Annual Sympsium on High-Performance Interconnects (HOTI)*, pages 79–82, 2013.

[22] K. H. Tsoi and W. Luk . Axel: A Heterogeneous Cluster with FPGAs and GPUs . In *International Symposium on Field Programmable Gate Arrays*, pages 115–124, 2010.

[23] Michael Showerman, Jeremy Enos, Avneesh Pant, et.al . QP : A Heterogeneous Multi-Accelerator Cluster . In *LCI International Conference on High-Performance Clustered Computing*, pages 54–57, 2009.

[24] R Ammendola, A Biagioni, O Prezza, F Lo Cicero, A Lonardo, P S Paolucci, D Rossetti, A Salamon, G Salina, F Simula, L Tosoratto, and P Vicini. Apenet+: high bandwidth 3d torus direct network for petaflops scale commodity clusters. *Journal of Physics: Conference Series*, page 042059, 2011.

[25] Mellanox Products Mellanox OFED GPUDirect RDMA Beta . http://www.mellanox.com/page/products_dyn?product_family=116&mtag=gpudirect .

[26] Stratix 10 - Overview . https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.highResolutionDisplay.html .

[27] Virtex-6 FPGA Configurable LogicBlock User Guide UG364(v1.2) . http://www.xilinx.com/support/documentation/user_guides/ug364.pdf .

[28] Home :: OpenCores . http://opencores.com .

[29] OpenCV port Projects RocketBoards.org . http://www.rocketboards.org/foswiki/Projects/OpenCV-Port .

[30] HIPAcc The Heterogeneous Image Processing Acceleration Framework . http://hipacc-lang.org.

[31] The ZYNQ BOOK . http://www.zynqbook.com/ .

[32] Open MPI: Open Source High Performance Computing . http://www.open-mpi.org .

[33] MVAPICH :: Home . http://mvapich.cse.ohio-state.edu .

[34] Intel MPI Library . https://software.intel.com/en-us/intel-mpi-library .

[35] Vivado Design Suite - Xilinx . http://japan.xilinx.com/products/design-tools/vivado.html .

[36] GPU-Accelerated Libraries . https://developer.nvidia.com/-gpu-accelerated-libraries .

[37] GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF) . https://gcc.gnu.org/ .

[38] The LLVM Compiler Infrastructure Project . http://llvm.org/ .

[39] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman . *Compilers: Principles, Techniques, and Tools* . Pearson Education, Inc, 1986.

[40] F.Vahid, G.Stitt, and R.Lysecky, . Warp Processing: Dynamic Translation . In *IEEE Computer*, pages 40–46, 2008.

[41] N.Clark, M.Kudlur, H.Park, S.Mahlke, and K.Flautner . Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization . In *International Symposium on Microarchitecture (MICRO 37)*, pages 30–40, 2004.

[42] Pin - A Dynamic Binary Instrumentation Tool, Intel Developer Zone . https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool .

[43] dlfcn.h dynamic linking . pubs.opengroup.org/onlinepubs/007904-975/basedefs/dlfcn.h.html .

[44] A. Nukada, K. Sato, and S. Matsuoka. Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[45] Ian Foster . *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* . Addison-Wesley Longman Publishing Co., Inc., 1995.

[46] Man page of PTHREADS . http://linuxjm.sourceforge.jp/html/LDP_manpages/man7/pthreads.7.html .

[47] boost C++ libraries, Chapter 30. Thread 4.1.0 - 1.54.0 . http://www.boost.org/doc/libs/1_54_0/doc/html/thread.html .

[48] OpenMP.org . http://openmp.org/wp .

[49] glibmm, Glib::Thread Class Reference . https://developer.gnome.org/glibmm/2.35/classGli_1_1-Thread.html .

[50] Shigang Li, Shucai Yao, Haohu He, Lili Sun, Yi Chen, and Yunfeng Peng. Extending synchronization constructs in openmp to exploit pipeline parallelism on heterogeneous multi-core. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 54–63, 2011.

[51] M. Gonzalez, E. Ayguadfi, X. Martorell, and J. Labarta. Defining and supporting pipelined executions in openmp. In *OpenMP Shared Memory Parallel Programming*. Springer Berlin Heidelberg, 2001.

[52] S. MacDonald, D. Szafron, and Jonathan Schaeffer. Rethinking the pipeline as object-oriented states with transformations. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 12–21, 2004.

[53] Navneet Dalal and Bill Triggs . Histograms of Oriented Gradients for Human Detection . In *International Conference on Computer Vision & Pattern Recognition (CVPR)*, pages 886–893, 2005.

[54] Automatically Tuned Linear Algebra Software (ATLAS) . http://math-atlas.sourceforge.net .

[55] GNU Octave . https://www.gnu.org/software/octave .

[56] Power Spectral Density Estimates Using FFT - MATLAB & Simulink - MathWorks . http://jp.mathworks.com/help/sig-nal/ug/psd-estimate-using-fft.html?lang=en .

[57] FFTW Home Page . http://www.fftw.org .

[58] cuFFT . https://developer.nvidia.com/cuFFT .

[59] OpenCV (Open Source Computer Vision) . opencv.willowgarage.com/wiki .

[60] Joao Bispo, Nuno Paulino, Joao M. P. Cardoso, and Joao Canas Ferreira. From instruction traces to specialized reconfigurable arrays. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 386–391, 2011.

[61] A.C.S.Beck, M.B.Rutzig, G.Gaydadjiev, L.Carro . Transparent reconfigurable acceleration for heterogeneous embedded applications . In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pages 1208–1213, 2008.

[62] J. Berdajs and Z. Bosnić. Extending applications using an advanced approach to dll injection and api hooking. *Software: Practice and Experience*, pages 567–584, 2010.

[63] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, pages 32–39, 2007.

[64] ADM-XRC-5T2 Page : Functionality and Apps | Alpha Data . http://www.alpha-data.com/products.php?product=adm-xrc-5t2 .

[65] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 2012.

[66] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory-bandwidth. In *IEEE Transactions on Parallel and Distributed Systems*, pages 695–705, 2013.

[67] Xilinx AXI Reference Guide (UG761) . http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_kguide.pdf .

[68] dlopen . pubs.opengroup.org/onlinepubs/009695399/functions/dl-open.html .

[69] Jinja2 (The Python Template Engine) . http://jinja.pocoo.org/docs/dev .

[70] Hello World in GLSL Lighthouse3d.com . http://www.lighthouse3d.com/tutorials/glsl-tutorial/hello-world-in-glsl .

[71] glRotatef - OpenGL Reference Pages . http://www.opengl.org/sdk/docs/man2/xhtml/glRotate.xml .

Publications

Related Papers

Journal Papers

[1] T. Miyajima, D. Thomas, H. Amano, A Toolchain for Dynamic Function Off-load on CPU-FPGA Platforms, *IPSJ Special issue of "Students' and Young Researchers' Papers*, Vol.56, Number.3, May.2015

[2] T. Miyajima, D. Thomas, H. Amano, Courier: A Toolchain for Application Acceleration on Heterogeneous Platforms, *IPSJ Transaction of System LSI Design Methodology*, Vol.8, Aug.2015, (To be published)
TSLDM Best Paper Award

International Conference Papers

[3] T. Miyajima, M. Arai, H. Amano, An FPGA Implementation of Line-Based Architecture 2-D Discrete Wavelet Transform Using Impulse C, *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2010)*, Tsukuba, Japan, Jun.2010

[4] T. Miyajima, M. Arai, H. Amano, An FPGA Implementation of Face Angle Detection System for Automobile using Impulse C, *International Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing 2011*, Grenoble, France, Mar.2011

[5] T. Miyajima, D. Thomas, H. Amano, A Domain Specific Language and Toolchain for Runtime Binary Acceleration, *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2012)*, pp.175-181, Okinawa, Japan, Jun.2012

[6] T. Miyajima, D. Thomas, H. Amano, A Domain Specific Language and Toolchain for OpenCV Runtime Binary Acceleration using GPU, *International Conference on Networking and Computing (ICNC 2012)*, Okinawa, Japan, Dec.2012

[7] T. Miyajima, T. Kuhara, T. Hanawa, H. Amano, T. Boku, Task level pipelining with PEACH2: an FPGA switching fabric for high performance computing. *International Conference on Field Programmable Technologies (FPT 2013)*, pp.466-469, Kyoto, Japan, Dec.2013

[8] T. Miyajima, T. Kuhara, T. Hanawa, H. Amano, T. Boku, Task Level Pipelining on Multiple Accelerators via FPGA Switch. *The 12th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2013)*, Innsbruck, Austria, Feb.2014

[9] T. Miyajima, D. Thomas, H. Amano, An Automatic Mixed Software Hardware Pipeline Builder for CPU-FPGA Platforms, *International Workshop on FPGAs for Software Programmers (FSP 2014)*, Munich, Germany, Sep.2014

Domestic Conference Papers and Technical Reports

[10] T. Miyajima, M. Arai, H. Amano, FPGA Implementation of Discrete Wav-elet Transform Using Impulse C *IEICE Technical Reports (RECONF)*, 2009-60 pp.35-40, Kanagawa, Japan, Jan.2010 (In Japanese)

[11] T. Miyajima, M. Arai, H. Amano, An FPGA Implementation of Line-Based Architecture 2-Dimensional Discrete Wavelet Transform Using Impulse C, *IEICE Technical Reports (CPSY)*,2009-84 pp.147-152, Tokyo, Japan, Mar.2010 (In Japanese)
SLDM Student Presentation Award

[12] T. Miyajima, M. Arai, H. Amano, An FPGA Implementation of Face Detection Recognition System for automobile using Impulse C *IEICE Technical Reports (RECONF)*, 2010-51 pp.71-76, Fukuoka, Japan, Dec.2010 (In Japanese)

[13] T. Miyajima, M. Arai, H. Amano, Resource Sharing in FPGA and Implementation of Face-Angle Detection Algorithm using Impulse C *IEICE Technical Reports (RECONF)*, 2011-01, pp.01-06, Hokkaido, Japan, May.2011 (In Japanese)

[14] T. Miyajima, D. Thomas, H. Amano, A Domain Specific Language and Tool-chain for Runtime Binary Acceleration, *IEICE Technical Reports (RECONF)*, 2012-05, pp., Okinawa, Japan, May.2012 (In Japanese)

[15] T. Miyajima, D. Thomas, H. Amano, Study and Evaluation of Runtime Binary Acceleration Mechanism for OpenCV and GPU, *IEICE Technical Reports (CPSY)*, CPSY2012-37 pp.37-42, Tokyo, Japan, Oct.2012 (In Japanese)

[16] T. Miyajima, T. Kuhara, T. Hanawa, D. Thomas, H. Amano, Study of Runtime Binary Acceleration on TCA node, *IEICE Technical Reports (RECONF)*, Kouchi, Japan, May.2013 (In Japanese)
CPSY Student Presentation Award

[17] T. Miyajima, T. Kuhara, T. Hanawa, D. Thomas, H. Amano, A study of pipeline execution on PEACH2, *IEICE Technical Reports (RECONF)*, Ishikawa, Japan, Sep.2013 (In Japanese)

[18] T. Miyajima, D. Thomas, H. Amano, Building a Mixed Software Hardware Pipeline on CPU-FPGA Platforms *IEICE Technical Reports (RECONF)*, RECONF2014-27 pp.57-62, Hiroshima,

Japan, Sep.2014 (In Japanese)
RECONF Student Presentation Award

Other Papers

International Conference Papers

[19] R. Uno, N. Ozaki, M. Izawa, A. Tsusaka, T. Miyajima, H. Amano, A Speculative Gather System for Cool Mega-Array, *International Conference on Field Programmable Technologies (FPT 2013)*, Kyoto, Japan, Dec.2013

[20] T. Kuhara, T. Miyajima, M. Yoshimi, H. Amano, An FPGA Acceleration for the Kd-tree Search in Photon Mapping. *International Workshop on Applied Reconfigurable Computing (ARC 2013)*, California, U.S.A, Mar.2013

[21] D. Kugami, T. Miyajima, H. Amano, A circuit division method for High-Level synthesis on Multi-FPGA systems, *The 27th IEEE International Conference on Advanced Information Networking and Applications (AINA-2013)*, Barcelona, Spain, Mar.2013

[22] M. Hatto, T. Miyajima, H. Amano, An Automatic Code Optimization for High Level Synthesis, *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2012)*, Okinawa, Japan, Jun.2012

Domestic Conference Papers and Technical Reports

[23] Y. Kishimoto, T. Toi, T. Miyajima, H. Amano, Design and Implementation of Adaptive Viterbi Decoder using Dynamic Reconfigurable System STP Engine, *IEICE Technical Reports (RECONF)*, RECONF2011-38 pp.93-98, Nagoya, Japan, Sep.2011 (In Japanese)

[24] M. Hatto, T. Miyajima, H. Amano, The Automatic Code Optimization for High-Level Synthesis, *IEICE Technical Reports (CPSY)*, CPSY2012-3 pp. 13-18, Tokyo, Japan, Apr.2012 (In Japanese)

[25] D. Kugami, T. Miyajima, H. Amano, Implementation of the circuit division for High-Level Synthesis, *IEICE Technical Reports (CPSY)*, CPSY2012-18 pp.55-60, Tottori, Japan, Aug.2012 (In Japanese)

[26] R. Uno, N. Ozaki, M. Izawa, A. Tsusaka, T. Miyajima, H. Amano, Implementation of Speculative Gather System for CMA, *IEICE Technical Reports (RECONF)*, Kouchi, Japan, May.2013 (In Japanese)

[27] D. Kugami, T. Miyajima, H. Amano, A circuit division method for High-Level synthesis on Multi-FPGA systems in stream processing, *IEICE Technical Reports (CPSY)*, CPSY2012-18 pp.55-60, Tottori, Japan, Jan.2013 (In Japanese)

[28] T. Kuhara, T. Miyajima, T. Hanawa, H. Amano, T.Boku, A FPGA/GPU cooperation in nodes communication using PEACH2, *IEICE Technical Reports (RECONF)*, RECONF2013-62 pp.37-42, Kanagawa, Japan, Jan.2014 (In Japanese)

[29] N. Sugimoto, T. Miyajima, T. Kuhara, T. Mitsuishi, H. Amano, Artificial Intelligence of Blokus Duo on FPGA Using Cyber Work Bench, *IEICE Technical Reports (RECONF)*, RECONF2013-58 pp.13-18, Kanagawa, Japan, Jan.2014 (In Japanese)

[30] T. Mitsuishi, S. Nomura, T. Miyajima, J. Suzuki, Y. Hayashi, M. Suga, H. Amano, Accelerating Breadth First Search on GPU-BOX, *IEICE Technical Reports (CPSY)*, CPSY2013-108 pp.235-240, Okinawa, Japan, Mar.2014 (In Japanese)

[31] Y. Katsuta, T. Miyajima, S. Nomura, T. Kuhara, T. Hanawa, H. Amano, T.Boku, Accelerating Breadth First Search on Tightly Coupled Accelerator, *IEICE Technical Reports (CPSY)*, Tokyo, Japan, Apr.2014 (In Japanese)

[32] M. Hatto, T. Miyajima, H. Matsutani, H. Amano, Optimized HOG for database system *IEICE Technical Reports (RECONF)*, RECONF2014-3 pp.11-16, Fukushima, Japan, Jun.2014 (In Japanese)

[33] T. Kuhara, T. Miyajima, T. Hanawa, H. Amano, Evaluation and Implementation of the Calculation Feature to PEACH2 *IEICE Technical Reports (RECONF)*, RECONF2014-28 pp.63-68, Hiroshima, Japan, Sep.2014 (In Japanese)