A Design Framework for Mapping Dataflow Graphs onto Heterogeneous Multiprocessor Platforms

by

Shuoxin Lin

Research outcome report submitted to the Marshall Plan Foundation
via the Salzburg University of Applied Sciences, in fulfillment of the requirements
for the Marshall Plan Scholarship
2015

Table of Contents

1 Introduction 1

2 Related Work 4

3 Framework 6
- 3.1 Dataflow Models . . . 6
- 3.2 Heterogeneous Computing Platform . . . 8
- 3.3 Vectorization . . . 10
- 3.4 Scheduling . . . 12
- 3.5 Integrated Workflow . . . 14
  - 3.5.1 Application Graph Specification and Actor Implementation . . . 14
  - 3.5.2 Multithreaded Programming Models . . . 16
  - 3.5.3 Data Transfer . . . 17
  - 3.5.4 Code Generation . . . 18

4 Experiments 21
- 4.1 Experimental Setup . . . 22
- 4.2 Scheduling . . . 24

5 Conclusion 28

6 Acknowledgement 29

Bibliography 30

Abstract

Dataflow models are valuable tools for representing, analyzing, and synthesizing embedded systems. Heterogeneous computing platforms with multi-core CPU and Graphics Processing Units (GPUs) provide a low cost platform for high performance computations. In this report, we present a dataflow based automated design framework that incorporates analysis, optimization and synthesis tools for embedded systems. Our framework is capable of generating high-performance software from dataflow applications targeted on heterogeneous CPU-GPU platforms. This framework exploits task and data-level parallelism in the dataflow specification and automatically utilizes the heterogeneous platform for performance gain without the need for manual, platform and application specific optimization. We demonstrate the novel and useful capabilities of this framework through experiments on an adapted MP-Sched benchmark that is representative for a wide range of DSP applications.

Chapter 1

Introduction

Embedded systems are rapidly increasing in their complexity and capabilities. Driven by continuously growing demand for functionality and performance, many types of embedded systems now utilize heterogeneous multiprocessor platforms. Among a variety of available classes of heterogeneous platforms, multicore CPU-GPU platforms, which integrate central processing unit (CPU) and graphics processing unit (GPU) devices, have been shown to provide significant performance gains on a wide range of embedded applications. An example of a widely-used CPU-GPU product family is the NVIDIA Tegra.

GPUs accelerate computational tasks by supporting data-level parallelism on a large scale with hundreds or thousands of SIMD (single instruction multiple data) multiprocessors. Achieving maximal performance gain of computational tasks on CPU-GPU platforms typically involves highly specialized optimization techniques that are specific to the application functions and hardware. In the context of embedded system development, where applications are often required to meet multidimensional constraints (e.g., constraints on throughput, latency, memory requirements, and power consumption), derivation and application of such specialized optimization techniques by hand can be highly error-prone and time-consuming. Moreover, designers sometimes need to migrate embedded software across platforms — for ex-

ample, to derive different versions of a given application with different trade-offs, or to utilize newer platform generations. As processing platforms continue to evolve, hand optimization techniques become increasingly difficult to track and maintain across platforms in an efficient and reliable manner.

Model-based design methodologies using dataflow models of computation have significant potential to help address the challenges in developing efficient and reliable GPU-CPU implementations with high productivity. Dataflow methods have been widely adopted in many application areas of embedded signal processing [1]. When using dataflow techniques in this context, the designer specifies an application as a directed graph, where vertices (*actors*) represent computational functions and edges represent inter-actor communication channels.

Dataflow formalisms provide well-structured and formally-rooted approaches for representing and implementing signal processing applications. Dataflow models allow important forms of design analysis and optimization to be applied systematically, including task scheduling, memory allocation and power management. Such capabilities provide the designer with valuable insight on implementation trade-offs that are difficult or impossible to extract purely from lower level representations, such as C or CUDA programs. Additionally, their high level of abstraction and systematic methods for component integration facilitate retargetable design methodologies, enabling efficient migration across diverse computing platforms, such as programmable digital signal processors, field programmable gate arrays, GPUs, and hybrid CPU-GPU systems.

An important challenge in advancing the state-of-the-art in dataflow-based

design methods is the automated synthesis of efficient embedded implementations on heterogeneous CPU-GPU platforms. Such automated synthesis needs to take into account complex implementation aspects that include actor-level vectorization, inter-processor data transfer, task scheduling, and dataflow buffer management. Development of dataflow design frameworks that systematically addresses these issues in CPU-GPU implementation is an important research area in high-performance embedded signal processing.

With this motivation, we have developed in this research a novel dataflow-based design framework that integrates relevant methods for analysis, optimization and synthesis of signal processing implementations on CPU-GPU platforms. Our framework is based on the Dataflow Interchange Format (DIF) [2], a standard language for representing dataflow models of signal processing applications, and the the Lightweight Dataflow Environment (LIDE) [3], which provides a programming methodology for implementing dataflow graph actors and edges in a wide variety of lower level languages. Our framework allows the designer to specify a signal processing application using dataflow models, and generate executable software that is optimized for efficient operation on a targeted CPU-GPU platform. Our framework also provides systematic exploration of design trade-offs on target platforms involving multidimensional design evaluation metrics. In this report, we focus specifically on demonstrating the capabilities of our framework in optimizing trade-offs between signal processing latency and throughput.

Chapter 2

Related Work

A variety of model-based design frameworks has been explored previously for heterogeneous multiprocessor platforms. For example, StreamIt [4], a popular dataflow-centered language and design infrastructure, has been applied to generate throughput-efficient software for GPU execution [5, 6]. These works focus on throughput optimization techniques for GPU kernel functions, considering optimized methods for memory coalescing, register allocation, and GPU utilization. StarPU [7] is a run-time task graph scheduling system for heterogeneous multiprocessor architectures. This system allows the designer to specify applications as task graphs, and perform run-time task scheduling for a heterogeneous platform with multi-core CPUs and a GPU.

Software optimization from dataflow models targeted to heterogeneous architectures has been studied in two different directions. The first direction focuses on efficient code generation for GPU kernels from fine-grained dataflow graphs [5, 6]. The second focuses on system level code synthesis for mix-grained dataflow graphs [8, 9]. Our work is related to this second direction. We go beyond the previous works in this direction by addressing problems that are critical to the performance of multicore signal processing systems, including graph-level vectorization, inter-processor data transfer, and CPU-GPU parallel execution.

In summary, the distinguishing aspect of the design framework that we present in this report is its integrated consideration of vectorization, heterogeneous multiprocessor scheduling, and optimized software synthesis for hybrid CPU-GPU platforms. The support in our design framework for vectorization allows data-level parallelism to be expressed in dataflow models and harnessed systematically by the GPU for parallel execution. Our approach to heterogeneous multiprocessor scheduling allows inter-actor (task-level) parallelism to be exploited in conjunction with our vectorization techniques for data-level parallelism. The software synthesis capabilities of our framework provide a high level of automation for the developed modeling and optimization techniques, allowing them to be applied with fast turnaround time.

Chapter 3

Framework

Our new dataflow-based design framework, which we refer to as the DIF-GPU framework (or simply "DIF-GPU"), aims to integrate important aspects of mapping signal processing applications onto heterogeneous platforms, including vectorization, scheduling and code generation, as depicted in Figure 3.1. Before discussing DIF-GPU in further detail, we present some relevant background on dataflow models.

3.1 Dataflow Models

A dataflow graph is a directed graph $G = (V, E)$ composed of a set of vertices V and a set of edges E. An *actor* $v \in V$ represents a computational task of arbitrary complexity. An edge $e = (u, v) \in E$ connects actors u to v, and represents a data buffer that stores *tokens* as they are communicated from the output of actor u to the input of v. Tokens represent the basic unit of data that is processed by actors. We define $u = src(e)$ as the the source actor of edge e, and $v = snk(e)$ as the sink actor of e. Dataflow actors are executed in terms of discrete units of execution, called *firings* of the associated actors.

Synchronous dataflow (SDF) is a specialized form of dataflow in which the numbers of tokens produced by an actor onto each output edge and consumed from each input edge are constant across all firings of the actor [10]. SDF is used widely

in the design and implementation of signal processing systems (e.g., see [1]). An important feature of properly-constructed SDF graphs is that they can be executed indefinitely (e.g., on unbounded streams of input data) with bounded memory requirements, which is an important feature for signal processing systems [10]. Such bounded memory execution can be achieved using a scheduling construct called a *valid periodic schedule* or simply *valid schedule*. SDF graphs for which valid schedules exists are called *consistent SDF graphs*.

For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the minimum number of firings of v in a valid schedule. The vector q of these repetition counts, indexed by the actors in the associated SDF graph, is called the *repetitions vector* of the graph. A variety of more general forms of dataflow has been proposed as alternatives to the SDF model. A few examples of such alternative models are Multidimensional SDF [11], parameterized dataflow [12], and Core Functional Dataflow [13]. DIF-GPU assumes that the input signal processing application is specified in terms of the SDF model. Extension of our framework to more general models, such as those listed above, is an interesting direction for future work.

Additionally, the input SDF graph is assumed to be acyclic and delay-free. – that is, the SDF graph does not contain any cyclic paths, and all edges have zero delay. Here, by a *delay*, we mean an initial token on the edge. Acyclic, delay-free SDF graphs can be used to represent a broad class of signal processing applications.

As described above, each edge e in an SDF graph is associated with a constant *production rate* and *consumption rate*, where these rates are in terms of tokens per

actor firing. These rates are denoted, respectively, as $prd(e)$ and $cns(e)$.

3.2 Heterogeneous Computing Platform

In this section, we describe the class of processing platforms that is targeted by our design framework. Heterogeneous computing platforms (HCPs) consist of multiple processor types, such as the hybrid CPU-GPU architectures targeted in this report. More specifically, our design framework focuses on an important class of cooperating single-GPP, single-GPU pairs. Each platform in this class consists of a multicore, general purpose processor (GPP) that is integrated with a GPU. The GPP, the main memory and the GPUs are connected via a shared bus, and the GPP controls overall execution flow, and is thus referred to as the "host processor" of the enclosing heterogeneous multiprocessor platform. The GPU receives instructions and data from the GPP, and is referred to as the (acceleration) device. In the remainder of this report, by an *HCP*, we mean a platform belonging to this class of single-GPP, single-GPU platforms that is targeted by our proposed new design framework.

In HCP architectures, GPUs are powerful Single Instruction Multiple Thread (SIMT) computational engines consisting of hundreds or thousands of processing cores that can concurrently perform computational tasks on massive data sets. GPU software is implemented with specialized programming models that facilitate exploitation of data-level parallelism. Parallel software for GPUs is written in terms of code modules called *kernels*. It has been shown that GPUs can achieve large

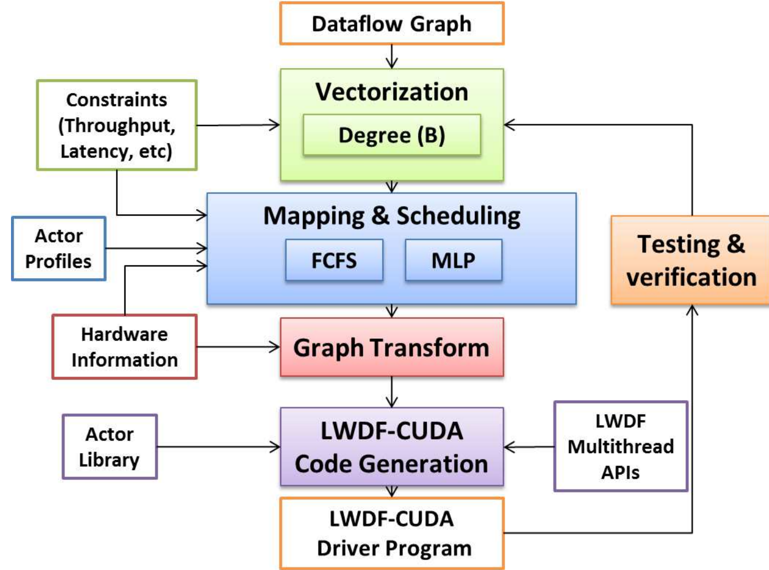


Figure 3.1: An illustration of the DIF-GPU framework for mapping signal processing applications onto heterogeneous platforms.

speed ups on some applications, but the realized performance gain varies greatly depending on the application, data set size and other factors.

Each GPU has its own memory (device memory), which is separated from main memory and other device memory. When data required for a GPU task in an HCP is outside the device memory, the GPU needs to copy the data into device memory using the shared bus. Data transfers between the host and the device are referred to as *host-to-device* or *device-to-host* data transfers depending on the direction. These data transfer steps result in large overhead that can significantly reduce the performance gain of HCPs [14]. In DIF-GPU, this issue is addressed in the scheduling step, where HCP data transfers are carefully modeled and optimized. The scheduling step is discussed further in Section 3.4.

3.3 Vectorization

Dataflow graph vectorization is a graph transformation that groups together multiple firings of a given actor into a single unit of execution [15, 16]. The number of firings involved in such a group is referred to as the *vectorization degree*. For example, if vectorization is applied to actor A with vectorization degree n, then blocks of n firings of A are executed together (sequentially on a single processor or concurrently across multiple processors).

In DIF-GPU, we apply a form of vectorization at the *graph level* in addition to the actor-level form of vectorization described above. The amount of graph-level vectorization applied is in general a positive integer, which is referred to as the *graph-level vectorization degree* (*GVD*). Use of a GVD in scheduling that is greater than 1 implies scheduling an unfolded version of the input dataflow graph [17]. This is a specialized form of unfolded scheduling where successive executions of individual actors are constrained to execute in blocks, as determined by the GVD. The vectorization degree of a given actor v in the input dataflow graph is given as $q(v) \times b$, where b is the GVD.

Let b be a GVD that is applied to an SDF graph $G = (V, E)$ in DIF-GPU. Then we derive another SDF graph $\nu_b(G)$, called the the b-vectorized graph of G. The b-vectorized graph may also be referred to simply as the *vectorized graph*. In $\nu_b(G)$, the actors and edges are in one-to-one correspondence with the actors and edges in G, respectively. Each actor v in $\nu_b(G)$ represents a vectorized version of the corresponding actor in G with vectorization degree $b \times q(v)$, where q is the

repetitions vector of G. Accordingly, the dataflow rate (production or consumption rate) associated with each actor port in $\nu_b(G)$ is b times the dataflow rate of the corresponding actor port in G.

Figure 3.2 shows an example of graph-level vectorization. Note that the repetition count of any actor in a b-vectorized graph is unity, independently of the value of b. In other words, if r represents the repetitions vector of the b-vectorized graph of G, for some $b \geq 1$, then $r(v) = 1$ for every actor v in $\nu_b(G)$.

DIF-GPU applies graph-level vectorization for two reasons. First, the framework aims to improve dataflow application throughput, which in turn requires optimized application of data- and task-level parallelism. With graph-level vectorization on a GPU target, $b \times q(A)$ firings of an actor A can execute concurrently if dependencies (through use of state) among firings of A don't require serialization of some of the firings. Even if A is mapped onto a single core of a GPP, the vectorized execution of A reduces the rate of context switching, and can improve performance further due to enhanced processor pipeline utilization and memory access locality [15, 18].

Second, many scheduling methods exist for mapping signal processing task graphs onto multiprocessor systems (e.g., see [19]). Here, by a *task graph*, we mean an acyclic dataflow graph in which all actors are fired at the same average rate. Because $r(v) = 1$ for all actors v, as described above, $\nu_b(G)$ is in the form of a task graph, and is therefore compatible with the rich library of existing task graph methods. Our framework takes advantage of these methods for generating efficient vectorized schedules for the given dataflow application.

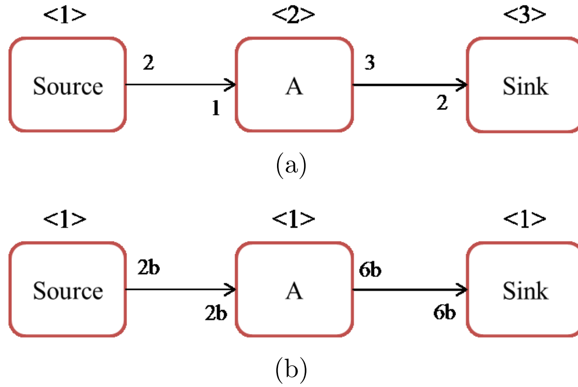


Figure 3.2: An illustration of graph-level vectorization. (a) Original SDF graph. (b) Vectorized SDF graph $\nu_b(G)$.

3.4 Scheduling

Dataflow scheduling for heterogeneous platforms is a complex problem. The problem is complicated by differences in actor execution times among different types of processors, and the overhead of interprocessor communication. Although finding optimal schedules in this context is NP-hard, a variety of heuristics has been developed.

In DIF-GPU, the scheduler takes the vectorized SDF graph $\nu_b(G)$ produced in the vectorization step, and generates a schedule for a single iteration of $\nu_b(G)$ as defined by the repetitions vector of $\nu_b(G)$. This schedule can then be iterated any number of times to provide an execution for the given DSP application.

As described previously, the vectorized graph is in the form of a task graph, and thus, various available task graph scheduling techniques can be applied in the DIF-GPU framework. Currently, we have incorporated two task graph scheduling techniques into DIF-GPU. We refer to these techniques as the First-Come-First-Serve (FCFS) and Mixed Linear Programming (MLP) schedulers.

FCFS is a well-known scheduling technique that is applicable in a wide variety of scheduling contexts. It has been studied previously in the context of CPU-GPU implementation by Teodoro et al. [20]. The FCFS scheduler in DIF-GPU manages a list of actors (the "ready list") that have sufficient data to be executed at any given time during scheduling. As the schedule evolves, the ready is list is updated. Whenever a processor is available, the scheduler assigns the actor with the shortest execution time in the ready list on that processor. This heuristic has the features of being simple and fast, but the quality of the schedules that it generates is often inferior to more sophisticated methods.

On the other hand, the MLP scheduler converts the task graph scheduling problem into a Mixed Linear Programming problem that can then be solved using off-of-the-shelf linear programming algorithms [16]. This method can generate efficient schedules, but can require a long running time to compute the schedule, especially when the input graph has large numbers of actors and edges. The FCFS and the MLP scheduler can be viewed as lying near two extremes of the trade-off between the schedule quality and scheduling speed — for this reason, they are interesting to start with to investigate design and implementation trade-offs using the DIF-GPU framework. Other scheduling heuristics, such as the Heterogeneous Earliest Finish Time (HEFT) [21] technique, can be integrated into the scheduling step of DIF-GPU to provide more trade-offs between schedule quality and scheduling speed. This is a useful area for further development of DIF-GPU.

3.5 Integrated Workflow

The DIF-GPU framework provides a complete dataflow graph scheduling and software synthesis workflow. The workflow encompasses application-level dataflow modeling (application graph specification), vectorization, scheduling, and code synthesis of cooperating C and CUDA subsystems for hybrid GPP/GPU implementation on the targeted HCP. In the remainder of this section, we discuss in more detail the different components of the DIF-GPU workflow.

3.5.1 Application Graph Specification and Actor Implementation

In DIF-GPU, dataflow models of DSP applications are specified using the Dataflow Interchange Format (DIF) [2] and the associated DIF package, a Java-based tool for specifying and analyzing dataflow models and applications. The vectorization and scheduling features of DIF-GPU as well as the code synthesis capabilities are implemented in and integrated into the DIF package.

The DIF package supports the DIF language, which is a language for specifying dataflow graph topologies. To implement the internal functionality of application graph actors, we employ the the lightweight dataflow environment (LIDE) [3], which provides retargetable application programming interfaces (APIs) for implementing dataflow actors in arbitrary platform-oriented languages, such as C, CUDA, and Verilog.

Developing an actor in LIDE requires implementation of four methods for the actor — namely *new*, *enable*, *invoke* and *terminate*. The new method performs

memory allocation and initialization for the actor. The enable methods returns a Boolean value indicating whether the actor is "fireable" — that is, whether sufficient data is available on its input edges, and sufficient empty space is available on its output edges when the method is called. The invoke method consumes input tokens from the actor input edges, performs the computation associated with the firing, and produces output tokens onto the actor output edges. LIDE does not place restrictions on the complexity of the invoke method. The terminate method frees memory that has been dynamically allocated for the actor.

LIDE supports various languages, including C, CUDA [22], and Verilog. In DIF-GPU, we use LIDE-C for actor implementation targeted to the GPP and LIDE-CUDA for actor implementation targeted to the NVIDIA GPU in our HCP platform. LIDE-C and LIDE-CUDA, are sub-packages within the LIDE package that support dataflow graph implementation in the C and CUDA languages, respectively. In DIF-GPU, we require that the actor implementations are vectorized. That is, each actor A should incorporate a positive-integer-valued vectorization parameter $vect(A)$, which specifies the number of successive firings of that are "treated as a single unit" for scheduling purposes. The enable and invoke functions for each actor A apply $vect(A)$ to, respectively, (a) check whether there is sufficient data and empty space available to support $vect(A)$ firings of $vect(A)$, and (b) execute $vect(A)$ firings of A.

3.5.2 Multithreaded Programming Models

DIF-GPU applies a multi-threaded programming model to implement actor scheduling on the targeted heterogeneous multi-processor platform. GPU and GPP operating systems typically provide extensive support for multithreading. Also, some operating systems provide methods to set the "affinity" of a task to a certain processor, which provides the programmer or software synthesis tool a means for guiding the mapping of program threads to processor cores. In DIF-GPU, we assume that each actor is implemented as a single thread. In the synthesized implementation, a total number of threads is created that is less than or equal to the number of processor cores. Furthermore, scheduling analysis within DIF-GPU assigns specific affinities to each actor's thread. That is, each actor's thread is assigned maximum affinity to a distinct processor core.

DIF-GPU employs a manager-worker thread model. Each GPP core and the GPU are treated as a individual "workers", and associated with individual worker threads. During run-time, a single manager thread keeps track of the state of the worker threads. When a worker thread is idle, the manager selects an actor and notifies the worker of the next actor to be fired. Upon receiving this notification, the worker starts executing the invoke method of the specified actor. When the invoke method completes, the worker notifies the manager thread, and the process repeats with selection and execution of the next actor by the manager thread.

DIF-GPU uses a self-timed scheduling approach for implementing schedules. The timing information used to construct the schedule is discarded before the code

generation phase, and only the sequence of vectorized actor firings on each processor (worker) is used. At runtime, the manager thread loads a pre-computed schedule in the application initialization phase. During schedule execution, the manager needs to periodically check the status of each worker (busy or idle) and the result from the enable method of the next actor in each worker's schedule. When a worker is idle and its next actor has sufficient data and empty space (as determined by its enable method), the manager can launch the actor to be executed by the worker. If all workers become busy at some point during execution, then the manager thread is blocked until an idle worker becomes available again.

3.5.3 Data Transfer

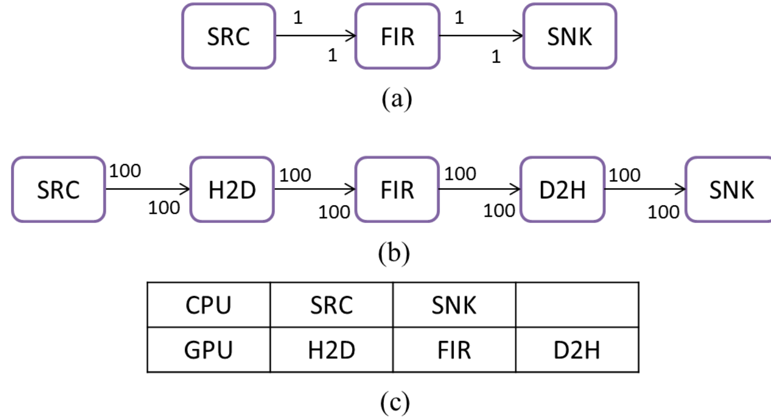
As described previously, a GPU has its own private memory space, called device memory. It has been shown that performing CPU-to-GPU and GPU-to-CPU transfers for each actor results in significant performance reduction [23], and therefore, should not be adopted in a practical framework. We apply the structure of the computed self-timed schedule in DIF-GPU to help eliminate unnecessary data transfers between the CPU and GPU. We do this by inserting special interprocessor communication (IPC) actors ("send" and "receive" actors) to provide the data transfers for each edge that has its source and sink actors mapped to different processors. This approach of incorporating send and receive actors is adapted from IPC modeling and implementation techniques discussed in [19].

3.5.4 Code Generation

DIF-GPU generates well-structured, human readable source code for compilation with back-end tools associated with the targeted HCP. Given a dataflow application graph G that is provided as input to DIF-GPU, we refer to the resulting synthesized software implementation as the *synthesized package* of G. The synthesized package contains a C++ header file (.h file), a C++ implementation file (.cpp file), and a set of schedule files, where each schedule file contains the schedule for a separate worker thread.

The C++ header and implementation files define a class that encapsulates the computation of G. Graph-level input, output and parameters can be applied through constructor arguments. In this manner, DIF-GPU generates an object-oriented module rather than generating a `main` function as the entry point for the derived executables. Through their modular structure, the implementations generated by DIF-GPU can be integrated flexibly into different design frameworks. This flexibility of integration is useful, for example, for generating DSP components in larger designs where it is not desired to employ dataflow techniques for all parts of the designs.

The synthesized package for an application graph depends on the LIDE package and and the library of LIDE-based actors that is used to construct the graph. Figure 3.3 shows a simple example based on a finite impulse response (FIR) filter. The application graph (Figure 3.3(a)) is transformed to the graph shown in Figure 3.3(b) after vectorization is performed and data transfer actors are inserted. The vectorization degree used in this example is $b = 100$. Figure 3.3(c) shows the



CPU	SRC	SNK	
GPU	H2D	FIR	D2H

Figure 3.3: An FIR filter example.

corresponding actor firing sequence on each processor when the GPU is used to accelerate the FIR filtering actor.

Figure 3.4 and Figure 3.5 show the synthesized code for the header and implementation files. The class `fir_graph_1` contains a constructor, a destructor, and a simple `execute` method. The constructor takes the data to be filtered and the filter length as two input parameters. The argument lists of the `execute` method and the destructor are empty. The constructor performs buffer allocation for the edges, initializes the graph actors, and initializes the manager-worker threads. The `execute` function starts the scheduler and provides the interface for executing the application graph. The destructor reclaims the memory allocated for the edges and actors.

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include "lide_c_actor.h"
#include "lide_c_fifo.h"
#include "lide_cuda_util.h"
#include "lide_cuda_fifo.h"
#include "lide_cuda_util.h"

        ⋮

#define H2D_0 1
#define ACTOR_FIR 2
#define ACTOR_SRC_1F 0
#define ACTOR_SNK_1F 4
#define D2H_0 3
#define ACTOR_COUNT 5
#define CPU 0
#define GPU 1

        ⋮

class fir_graph_1 {
public:
    fir_graph_1(float* fir_filter, int fir_ntaps);
    ~fir_graph_1( );
    void execute();
private:
    lide_cuda_thread_list* thread_list;
    lide_c_actor_context_type* actors[ACTOR_COUNT];
    char *descriptors[ACTOR_COUNT];
    lide_cuda_fifo_pointer edge_in_h2d_0;
    lide_cuda_fifo_pointer edge_out_h2d_0;
    lide_cuda_fifo_pointer edge_in_d2h_0;
    lide_cuda_fifo_pointer edge_out_d2h_0;
};
```


Figure 3.4: Header code for the FIR filter example.

```
#include "fir_graph_1.h"


fir_graph_1::fir_graph_1(float* fir_filter, int fir_ntaps)
{
    /* Full constructor */
    edge_in_h2d_0 = lide_cuda_fifo_new(512, sizeof(float), CPU);
    edge_out_h2d_0 = lide_cuda_fifo_new(512, sizeof(float), GPU);
    edge_in_d2h_0 = lide_cuda_fifo_new(512, sizeof(float), GPU);
    edge_out_d2h_0 = lide_cuda_fifo_new(512, sizeof(float), CPU);
    actors[ACTOR_SRC_1F] = (lide_c_actor_context_type*) lide_cuda_src_1f_new
          (edge_in_h2d_0, 512, 2);
    actors[ACTOR_FIR] = (lide_c_actor_context_type*) lide_cuda_fir_new(edge_out_h2d_0,
           edge_in_d2h_0, 512, 512, 2, fir_filter, fir_ntaps, GPU);
    actors[ACTOR_SNK_1F] = (lide_c_actor_context_type*) lide_cuda_snk_1f_new
          (edge_out_d2h_0, 512, 2);
    actors[H2D_0] = (lide_c_actor_context_type*) lide_cuda_memcpy_new(edge_in_h2d_0,
          edge_out_h2d_0, 512, 512);
    actors[D2H_0] = (lide_c_actor_context_type*) lide_cuda_memcpy_new(edge_in_d2h_0,
          edge_out_d2h_0, 512, 512);

    descriptors[ACTOR_SRC_1F] = "ACTOR_SRC_1F";
    descriptors[ACTOR_FIR] = "ACTOR_FIR";
    descriptors[ACTOR_SNK_1F] = "ACTOR_SNK_1F";
    descriptors[H2D_0] = "H2D_0";
    descriptors[D2H_0] = "D2H_0";
    char* file_names[1] = {"thread_0.txt"};
    thread_list = lide_cuda_thread_list_init_2(NUMBER_OF_THREADS, file_names, actors,
          ACTOR_COUNT, descriptors, 1);
}
void fir_graph_1::execute()
{
    lide_cuda_thread_list_scheduler(thread_list);
}

fir_graph_1::~fir_graph_1() {
    lide_cuda_thread_list_terminate(thread_list);

    lide_cuda_fifo_free(edge_in_h2d_0);
                ⋮
    lide_cuda_src_1f_terminate((lide_cuda_src_1f_context_type*)actors[ACTOR_SRC_1F]);
    lide_cuda_fir_terminate((lide_cuda_fir_context_type*)actors[ACTOR_FIR]);
    lide_cuda_snk_1f_terminate((lide_cuda_snk_1f_context_type*)actors[ACTOR_SNK_1F]);
    lide_cuda_memcpy_terminate((lide_cuda_memcpy_context_type*)actors[H2D_0]);
    lide_cuda_memcpy_terminate((lide_cuda_memcpy_context_type*)actors[D2H_0]);

}
```


Figure 3.5: Implementation code for the FIR filter application.

Chapter 4

Experiments

We have implemented first versions of the proposed DIF-GPU workflow, including the heterogeneous multiprocessor schedulers (FCFS and MLP), and code generator described in Section 3. In this section, we demonstrate this first version of the DIF-GPU framework by evaluating its performance on an adaptation of the MP-Sched benchmark [24], which is illustrated in Figure 4.1.

Our adapted benchmark, which we refer to as the *sliding-window inner product* (*SWIP*)) system, describes a signal processing flow graph that consists of a grid of $P \times S$ actors (called *SWIP actors*) that perform inner product computations on sliding windows of data. Here, P is the number of pipelines and S is the number of stages. The SWIP benchmark is relatively easy to construct as an initial benchmark, and involves computations that are representative of common computations in the DSP domain. Each SWIP actor consumes an array $x[n]$ of size L, performs a sliding window inner product function, and produces an output array $y[n]$ of size L. The computation performed by a SWIP actor can be expressed as follows:

$$y[n] = \sum_{k=-r}^{r} W[r+k]x[n+k]. \tag{4.1}$$

where $W[n]$ is a pre-defined window of length $2r+1$. The benchmark is a synthetic benchmark with a parameterized structure that is representative of a class of prac-

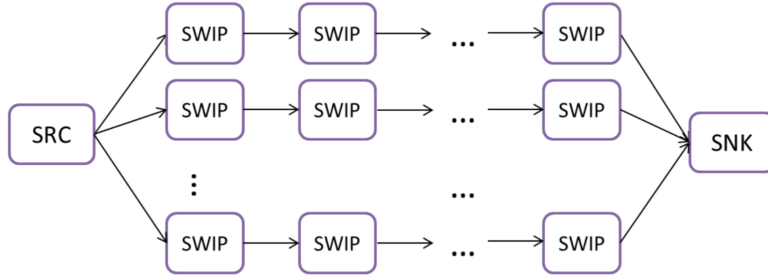


Figure 4.1: An illustration of the SWIP benchmark.

tical signal flowgraph structures. The benchmark is a non-trivial problem for the multiprocessor schedulers in DIF-GPU because of the multiple forms of parallelism that are employed, including fine grained parallelism within actors, task-level parallelism across different pipelines, and pipeline parallelism within a given pipeline.

4.1 Experimental Setup

We use an Intel Core i7-2600K CPU with an NVIDIA GeForce GTX680 GPU for our experiments. In our framework, the benchmark application graph is specified in the DIF language. Then the graph is vectorized over selected vectorization degrees. Here, we selected the following set of vectorization degrees:

$$B = \{b = 256k | k = 1, 2, ..., 10\}. \tag{4.2}$$

For each vectorized graph, DIF GPU computes its schedule and mapping based on the selected scheduling strategy. A C++ header file and implementation file that realize the dataflow application are then generated.

For each SWIP actor, we set the window length to be 7. The actors are

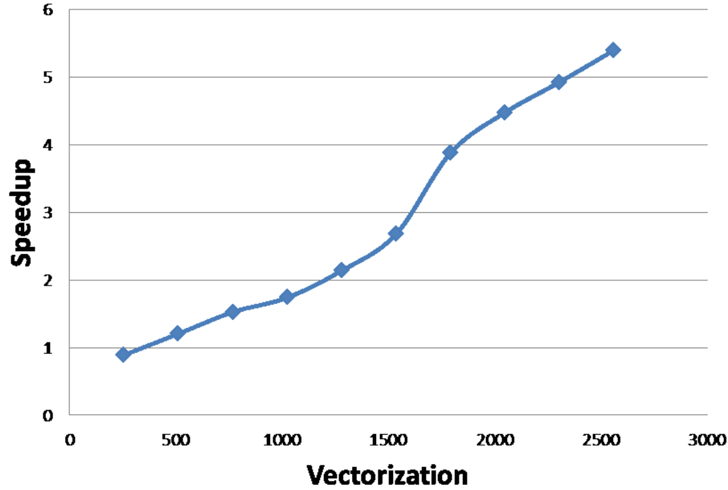


Figure 4.2: Speedup of a SWIP actor.

individually profiled for the specified vectorization degrees. The speedup of a GPU-accelerated SWIP actor is shown in Figure 4.2, excluding data transfer time. We observe an increase in speedup by increasing the vectorization degree for GPU-accelerated implementation. When $b = 256$, the actor runs slower when mapped onto a GPU because the GPU cores are significantly under-utilized, and a small number of GPU cores cannot provide performance gain over a powerful CPU. As the vectorization degree increases, more GPU-cores operate in parallel, providing more speedup. When b increases, the speedup continues to improve to 5.4x at $b = 2,560$.

In our experiments, we found that data transfer can be a significant overhead in mapping the the SWIP benchmark to the targeted HCP. Figure 4.3 shows a comparison between computation time and data transfer overhead for an individual SWIP actor. In this setting, the host-to-device and and device-to-host data transfer times are comparable or even higher than the actor computation times. This is due

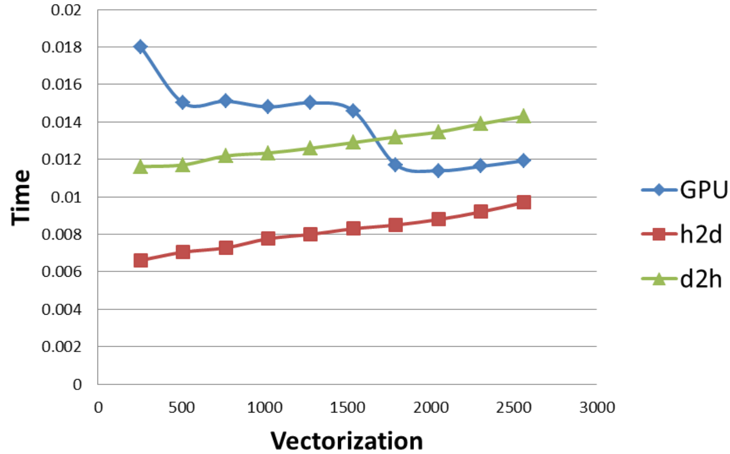


Figure 4.3: Comparison of the computation time of a SWIP actor, and H2D and D2H data transfers.

to the relatively small computational load of a SWIP actor. For the GTX680, there are $1,536$ CUDA cores, each of which is capable of executing computations on a warp (group of 32 CUDA threads), which enables a group of $41,952$ floating point numbers to be processed concurrently. At $b = 2,560$, the device is significantly under-utilized, and all $2,560$ tokens can be processed on the device concurrently. Therefore we see no performance increase with increases in b at this range of vectorization levels. The sum of the H2D and D2H times is 2.0 times more than the computational time when $b = 2,560$.

4.2 Scheduling

We compare the simulated throughput of the benchmark application graph using the two implemented schedulers (FCFS and MLP) for the 2x5 and 4x4 configurations of the SWIP benchmark. The simulated throughput is defined as (N/M), where M is the execution time of the generated schedule (i.e., of one iteration of

vectorized graph), and N is the number of tokens processed within that period.

Figure 4.4 shows the throughput for the 2x5 benchmark under 4 different mappings. The *Single-CPU* mapping assigns all actors to a single CPU core. *Single-GPU* maps all actors with GPU-accelerated implementations on the GPU and the rest on a single CPU core. *FCFS* maps the actors according to the schedule generated by the FCFS scheduler, where the target architecture contains 1 CPU core and 1 GPU. *MLP* maps the actors according to the schedule generated by the MLP scheduler, where the target architecture again contains 1 CPU core and 1 GPU. We observe that among the 4 mappings, the MLP mapping obtains the highest simulated throughput. It outperforms the single-GPU mapping by at least 20%. On the other hand, the FCFS mapping does not provide a consistent improvement over the single-GPU mapping. It outperforms the single-GPU mapping for small vectorization degrees, when the CPU performance of a SWIP actor is similar to the GPU performance. However, as the vectorization degree increases, FCFS mappings are not able to efficiently utilize the GPU in the 2x5 mpsched benchmark. In our experimental setup, the improvement quickly saturates as the vectorization degree increases beyond 1024.

Figure 4.5 shows the throughput for the 4x4 SWIP benchmark under 4 different mappings. Here, we use the FCFS scheduler with two target architectures: 1 CPU core + 1 GPU, and 3 CPU cores + 1 GPU. We do not provide MLP mappings here due to its very large schedule computation time [16]. We observe that among the 4 mappings, the 3 CPU + 1 GPU FCFS outperforms other mappings, as it utilizes more CPU cores for computation. This is especially significant when the

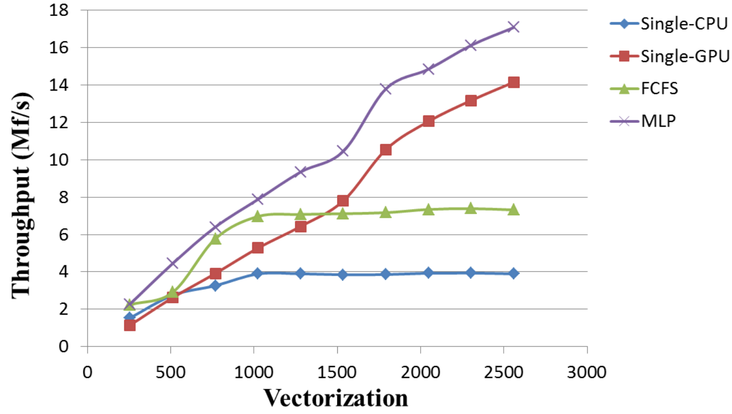


Figure 4.4: Simulated throughput on different mappings for the 2x5 SWIP benchmark.

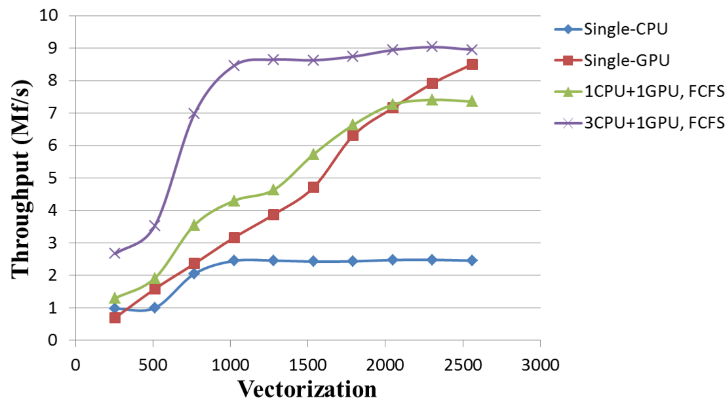


Figure 4.5: Simulated throughput on different mappings for the 4x4 SWIP benchmark.

vectorization degree is small (the GPU has less speedup over the CPU). As the GPU speedup grows, the relative improvement provided by the 3 CPU + 1 GPU FCFS decreases.

To evaluate the actual running time for the benchmark with different mappings, we implemented the 2x5 SWIP benchmark using the LIDE multithreading APIs. Figure 4.6 shows the actual running time of the application. Overall, the MLP provides the mapping with the highest throughput, but its improvement over a single-GPU mapping is smaller compared to the simulation results. The greatest

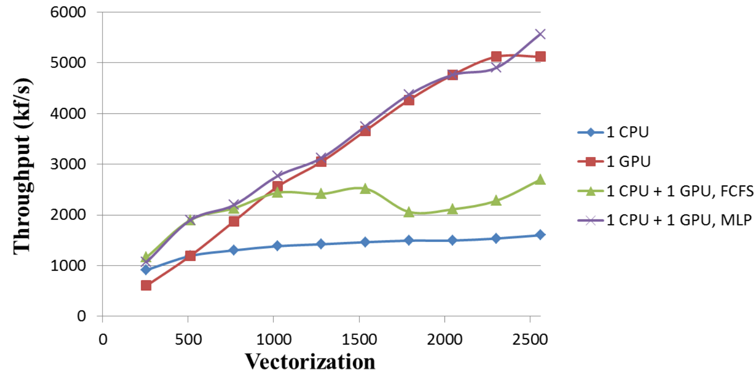


Figure 4.6: Actual throughput of the 2x5 SWIP benchmark.

improvement occurs for $v = 2,560$, This difference in performance may be due to the operating system overhead for multi-thread management. For small vectorization degrees ($v < 1,000$), the MLP and FCFS mappings perform similarly. This indicates that when the performance gain on the GPU is relatively small, utilizing more CPU cores with the FCFS scheduler may provide significant speed-up gain.

Chapter 5

Conclusion

In this report, we have presented a new integrated design framework, called DIF-GPU, that integrates graphics processing unit (GPU) acceleration in the dataflow interchange format (DIF), and targets multi-core heterogeneous computing platforms (HCPs) that combine central processing unit (CPU) and GPU devices. Through experiments, we have demonstrated the utility of DIF-GPU to facilitate efficient design space exploration, and enhance application performance by vectorizing the application graph, generating efficient schedules, reducing inter-processor communication overhead, and performing automated code generation. We have shown that for some application configurations, DIF-GPU is able to generate implementations that outperform conventional HCP mappings where all GPU-accelerated actors are mapped to a GPU. Useful directions for future work include: (1) further investigation of efficient mapping methods for CPU-GPU heterogeneous platforms; (2) evaluation of DIF-GPU on a broader range of benchmarks; and (3) extending DIF-GPU so that it can handle targets that involve multi-core CPUs and multiple GPU devices.

Chapter 6

Acknowledgement

This research was sponsored in part by the Austrian Marshall Plan Foundation, and the Laboratory for Telecommunication Sciences.

Bibliography

[1] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, second edition, 2013, ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[2] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[3] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, "A lightweight dataflow approach for design and implementation of SDR systems," in *Proceedings of the Wireless Innovation Conference and Product Exposition*, Washington DC, USA, November 2010, pp. 640–645.

[4] M. I. Gordon, W. Thies, and Saman Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.

[5] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2009, pp. 200–209.

[6] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 381–392.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Journal of Concurrency and Computation: Practice & Experience*, vol. 23, no. 2, pp. 187–198, February 2011.

[8] F. Ciccozzi, "Automatic synthesis of heterogeneous cpu-gpu embedded applications from a uml profile," in *Proceedings of the International Workshop on Model Based Architecting and Construction of Embedded Systems*, 2013.

[9] H. Jung, Y. Yi, and S. Ha, "Automatic CUDA code synthesis framework for multicore CPU and GPU architectures," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203 of *Lecture Notes in Computer Science*, pp. 579–588. Springer Berlin Heidelberg, 2012.

[10] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[11] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.

[12] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, June 2000, pp. 1948–1951.

[13] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[14] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[15] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[16] G. Zaki, W. Plishker, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio," *Journal of Signal Processing Systems*, vol. 70, no. 2, pp. 177–191, February 2013, DOI:10.1007/s11265-012-0696-0.

[17] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, Inc., 1999.

[18] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing for DSP software optimization," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 163–177, February 2008.

[19] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, CRC Press, second edition, 2009, ISBN:1420048015.

[20] G. Teodoro, R. Sachetto, O. Sertel, M. N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira, "Coordinating the use of gpu and cpu for improving performance of compute intensive applications," in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–10.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[22] *CUDA C Programming Guide*, September 2015, Version 7.5.

[23] P.-H. Horrein, C. Hennebert, and F. Pétrot, "Integration of GPU computing in a software radio environment," *Journal of Signal Processing Systems*, vol. 69, no. 1, pp. 55–65, 2012.

[24] E. Blossom, "GNU radio: tools for exploring the radio frequency spectrum," *Linux Journal*, June 2004.