Calypso : A GPU streaming framework. For Software Defined Radio implementations

Fotis Nikolaidis

Thesis submitted in partial fulfillment of the requirements for the

Masters’ of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Crete, Greece

Thesis Advisors: Prof. Apostolos Traganitis
Co-advisor: Dr. Stefanos Papadakis
University of Crete
Computer Science Department

Calypso: A GPU pipeline engine. For Software Defined Radio implementations

Thesis submitted by
Fotis Nikolaidis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:__________________________________________
Fotis Nikolaidis

Committee approvals:__________________________________________
Apostolos Traganitis
Professor, Thesis Supervisor

Evangelos Markatos
Professor, Committee Member

Panagiotis Tsakalides
Professor, Committee Member

Departmental approval:__________________________________________
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, November 2015
Abstract

Software Defined Radio devices have been proposed to solve the problem of difficult prototyping, update and upgrade of fixed-circuit radios.

They are radio communication systems in which the signal processing circuits have been replaced by software running on an external host. The exchanged information is in the form of digitized baseband signals. This makes the radio able to used regardless of the underlying signal processing system. Thus, it can be used with any telecommunication standard.

The fact that everything is written in software and executed in signal-agnostic radios, gives great boost to reconfigurable networks, efficiency in power spectrum usage, feature extensibility, bug-fixing and rapid prototyping.

Nevertheless, the fixed-circuit performance must be equalized by the software counter-part. During the previous decades, only specialized hardware like Field Programmable Gate Array and Digital Signal Processing could provide the necessary computational power.

Lately, that General Purpose Processor have become more powerful, they are preferred over specialized hardware for programmability reasons. To further increase the computational power without losing the programmability benefits, it has been proposed to put into service the Graphics Processing Units (GPUs), for signal processing purposes. However existing signal-processing engines have been built for CPU environments, and any effort for GPU integration is constrained by the core design.

This dissertation presents Calypso: a hybrid batch streaming engine designed to take advantage both of Central Processing Units (CPUs) and GPUs. It is constituted of self-contained entities, called modules. Modules have an upper CPU layer (orchestrator) emphasizing on task parallelisation and a lower GPU layer (labor) emphasizing on data parallelisation.

The labor layer provides the functionality and must be implemented by the developer. The upper layer belongs to Calypso and is responsible for operational duties like: task synchronization, ordering, data passing, asynchronous operations and memory management. With this toolkit, even an amateur developer in parallel programming can build asynchronous applications with GPU support.

To evaluate the engine, the Digital Video Broadcasting Terrestrial (DVB-T) has been implemented in CPU, as GNU Radio application, and in GPU, as Calypso application. Measurements shown that Calypso performs twice as better than GNU Radio for a single DVB-T stream, on a much cheaper hardware. However the technological gap between the commercial GPU and the high-end CPU becomes evident when multiple DVB-T are concurrently active. The GPU is saturated faster that the CPU, which results to equal an distribution of the aggregated throughput to the streams.
Περίληψη

Οι συσκευές προγραμματιζόμενες σε λογισμικό (Software Defined Radios) έχουν προταθεί προκειμένου να λύσουν το πρόβλημα της δύσκολης προτύποποιήσης, αναβάθμισης και εξυγχρόνισης των κλασσικών τηλεπικοινωνιακών συσκευών. Στις συσκευές αυτές τα κυκλώματα επεξεργασίας σήματος έχουν αντικατασταθεί από λογισμικό το οποίο εκτείλεται σε ένα εξωτερικό μηχάνημα, με το οποίο ανταλλάσσει ψηφιοποιημένα σήματα. Λειτουργούν ανεξάρτητα από το εκάστοτε σήμα, με αποτέλεσμα να υποστηρίζουν οποιοδήποτε τηλεπικοινωνιακό πρότυπο. Ανάλογα με το λογισμικό αλλάζει και η υποστήριξη της συμπεριφοράς τους. Η ιδιαίτερότητα αυτή δίνει μεγάλη ώθηση στα "έξυπνα' και προσαρμοζόμενα δίκτυα, στην διόρθωση σφαλμάτων και στην εύκολη προσθήκη νέων λειτουργιών.

Ωστόσο, η επίδοση του λογισμικού πρέπει να είναι εφάμιλλη με αυτή των ηλεκτρονικών κυκλωμάτων. Προς αυτή την κατεύθυνση, έχει χρησιμοποιηθεί εξειδικευμένος υλικό όπως Field Programmable Gate Array και Digital Signal Processing, στο οποίο εκτελούνται οι τηλεπικοινωνιακές εφαρμογές. Τελευταία, οι επεξεργαστές γενικού σκοπού προτιμούνται σε σχέση με τις πιο εξειδικευμένες λύσεις, αν και υποστηρίζουν σε απόδοση, λόγω της ευκολίας προγραμματισμού που παρέχουν. Για την αντιστάθμιση της απόδοσης, έχει προταθεί η χρήση των Graphics Processing Unit για την επεξεργασία των σήματος. Οι σημερινές μηχανές επεξεργασίας σήματος είναι σχεδιασμένες για περιβάλλοντα Central Processing Unit. Αν και έχουν υπάρξει προσπάθειες ενσωμάτωσης με Graphics Processing Unit (GPU), είναι ελάχιστες και περιορισμένες λόγω των κληρονομικών χαρακτηριστικών της Central Processing Unit (CPU).

Αυτή η εργασία παρουσιάζει μια υβριδική μηχανή επεξεργασίας σήματος με το όνομα Calypso. Αποτελείται από ένα κομμάτι που εκτελείται στην CPU και αποσκοπεί στην παραλληλοποίηση των διεργασιών και ένα κομμάτι που εκτελείται στην GPU και αποσκοπεί στην παραλληλοποίηση των δεδομένων. Ο κώδικας της GPU περικλύεται σε μία οντότητα που εκτελείται στην CPU και ονομάζεται module. Κάθε module εκτελείται ασύγχρονα, ενώ σε συνδυασμό με άλλα modules σχηματίζεται μία γραμμή επεξεργασίας. Η Calypso χειρίζεται αυτόματα την παραλληλοποίηση των διεργασιών, την μεταφορά των δεδομένων, τις ασύγχρονες πράξεις και την διαχείριση της μνήμης. Ως εκ τούτου, ακόμα και ένας αρχάριος προγραμματιστής μπορεί να υλοποιήσει μία παράλληλη εφαρμογή, με υποστήριξη GPU, χωρίς κόπο.

Για την αξιολόγηση του συστήματος έχει υλοποιηθεί το πρότυπο ψηφιακής μετάδοσης επίγειας τηλεόρασης σε CPU και σε GPU. Για την υλοποίηση σε CPU χρησιμοποιήθηκε το GNU Radio ενώ για την υλοποίηση σε GPU η Calypso. Αποδεικνύεται πως χρησιμοποιώντας την Calypso η συνολική απόδοση είναι δύο φορές καλύτερη από τον GNU Radio. Ωστόσο, λόγω των πιο εξειδικευμένων τεχνολογικών CPUs σε σχέση με την εμπορική GPU, οταν υπάρχουν πολλαπλές ροές η GPU συμφωνείται πιο γρήγορα από την CPU, με αποτέλεσμα η συνολική απόδοση να μοιράζεται ηξίου σε όλες τις ροές.
Acknowledgements

For the completion of this thesis, firstly I would like to thank the Foundation of research and technology Hellas for the provided hardware and the University of Crete for the software skills it has given me all these years.

As for the liveware, I would like to thank my supervisors Apostolos Traganitis and Stefanos Papadakis for their full cooperation. Manolis Surligas and Antonis Makrogiannakis for their help on technical issues. Debbie Maliotakiis and Vasileios Koliais for their seems-like-English review for versions 3, 4 and 7 (Currently we are in version 10).

Maria Keramarou, Christoforos Koutelieris, Giannis Meros, Christos Tzagkarakis and all the people who have accompanied the raki all these years. Captain Dimitris Politis for his task redistribution skills, without whom the thesis would be still at version 4.

Further, I would like to thank the secretary of the department, Gkeli Kosma for her helpfulness in administrative issues over this long period. Not to forget each and every fellow who has attended the fiasco-presentation.

Finally and heartily I would like to thank Manolis Ploumidis for all the girls he has brought and Petros Gkigkis for hiding the corpses into Nick Kossifidis’ place.....
## Contents

1 Introduction .......................................................... 2  
   1.1 Software Defined Radios ........................................ 2  
      1.1.1 General Purpose processors ............................... 3  
      1.1.2 GP-GPU history ........................................... 4  
   1.2 Motivation .......................................................... 5  
   1.3 Related work .......................................................... 5  
   1.4 Calypso Contribution .............................................. 6

2 Calypso ................................................................. 7  
   2.1 Features ............................................................ 7  
   2.2 Module ............................................................. 9  
   2.3 Memory Manager .................................................. 11  
   2.4 Signature .......................................................... 14  
      2.4.1 Outbound Fractional ....................................... 15  
      2.4.2 Inbound Fractional ......................................... 15  
   2.5 Topologies .......................................................... 17  
      2.5.1 Slow module scenario .................................... 18  
   2.6 Deeper into components ......................................... 18  
      2.6.1 Memory Functions ......................................... 18  
      2.6.2 Classes ..................................................... 18  
   2.7 Interface ............................................................ 20  
   2.8 Gnuradio vs Calypso .............................................. 22

3 Optimisation ............................................................ 23  
   3.1 Algorithmic ....................................................... 23  
      3.1.1 Parallelization levels .................................... 23  
      3.1.2 Lookup ..................................................... 23  
   3.2 Software Optimization .......................................... 24  
      3.2.1 Memory Alignment ........................................ 24  
      3.2.2 Cache Coherence .......................................... 24  
      3.2.3 Structs and Arrays ....................................... 24  
   3.3 Effective coding .................................................. 25  
   3.4 Hardware optimization ........................................... 26  
      3.4.1 Scalar vs Vector processors ............................. 26  
   3.5 NVIDIA CUDA ...................................................... 26  
      3.5.1 Hardware Arrangement .................................... 27  
      3.5.2 indexing ................................................... 27  
      3.5.3 Memory Hierarchy ......................................... 28
Chapter 1

Introduction

1.1 Software Defined Radios

In the history of mankind, humans have invented multiple ways to communicate from distance. The oldest reference is about a system in Ancient Greece, composed by large torches on top of hills. This system allowed the Greeks to send the message for the fall of the Troia back to Mycenes over a night instead of week that would take otherwise.

"Visual Telecommunication" means like fires and flags were dominant until the 17th century. When the electromagnetism was comprehended and manipulated, the era of "Electromagnetic Telecommunications" begun.

As the electronics advanced, devices became smaller and powerful. Nowadays, they have been miniaturized that much, that they have become essential part of the daily life.

Nevertheless, from fryctories to contemporary radio-on-chip, the flaw is common; they are built on hardware. Once they are built, it is impossible to reconfigure, customize or upgrade them without replacement.

At the beginning of 1970, a researcher at a DoD laboratory promoted the idea of "digital receivers" which later renamed to "Software defined radios" (Fig. 1.1) [1][2]. He suggested a device in which the digital signal processing circuits are replaced by software implementations executed on external devices. The external device interacts with the Software Defined Radio (SDR) by exchanging digitized baseband signals.

Only a minimal subset of circuits which cannot be implemented in software, remain within the SDR device. Specifically: a) RF-frontend for analog signal processing (filters, frequency synthesizers and up/down convertors), b) analog to digital convertors and c) digital frontend used for equalization, interference cancellatation and communication with the external device.

The benefits of having a generic signal-agnostic device that can "speak" any Telecommunication standard range from efficient spectrum usage to rapid prototyping of new standards without the delay and the cost of printing dedicated circuits.

However, this comes with the cost of equalizing the software performance to that of a dedicated circuit. When General Purpose Processors (GPPs) were too slow for signal processing, the software was executed on specialized hardware like Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) (Fig. 1.3). As GPPs became faster, they are preferred over more specialized solution due to the programmatibility they provide.
1.1.1 General Purpose processors

The two dominant types of GPPs are the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). Table 1.1 compares the features of the CPU and the GPU of a modern, off-the-shelf workstation as in the year 2015.

The GPU attributes almost 20 times more Giga flops than the CPU, at a clock rate significantly lower. To achieve that, in GPUs the control logic circuits have been replaced
CHAPTER 1. INTRODUCTION

Table 1.1: Comparison table of commodity CPU and GPU features

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Intel Xeon E5-2630 v2</td>
<td>NVIDIA QUADRO K2000</td>
</tr>
<tr>
<td>Computation</td>
<td>35.7 Gflops</td>
<td>723 Gflops</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.6 GHz</td>
<td>954 MHz</td>
</tr>
<tr>
<td>Cores</td>
<td>12 (Logical)</td>
<td>384 SMX</td>
</tr>
<tr>
<td>Consumption</td>
<td>65 W</td>
<td>51 W</td>
</tr>
<tr>
<td>Price</td>
<td>700 Euro</td>
<td>400 Euro</td>
</tr>
</tbody>
</table>

by many, smaller Arithmetic logic units (ALUs) [3]. Therefore the GPUs can do large amounts of bulky mathematical labor but cannot be used efficiently when it comes to logical operations.

Signal processing applications can take advantage of the provided capabilities like a) performance and precision in arithmetic operations b) efficiency in matrix manipulation and c) multi-level parallelism. The use of GPUs for purposes other than video processing is called General-purpose computing on graphics processing units (GPGPUs).

1.1.2 GP-GPU history

General-purpose computing [4] [5] on GPUs only became practical and popular after 2001, with the advent of both programmable shaders and floating point support on graphics processors. In particular, problems involving matrices and/or vectors - especially two-, three-, or four-dimensional vectors - were easy to translate to a GPU, which acts with native speed and support on those types. The scientific computing community’s experiments with the new hardware started with a matrix multiplication routine.

These early efforts to use GPUs as general-purpose processors required reformulating computational problems in terms of graphics primitives, as supported by the two major APIs for graphics processors, OpenGL and DirectX. This cumbersome translation was obviated by the advent of general-purpose programming languages and APIs such as Sh/RapidMind, Brook and Accelerator.

These were followed by Nvidia’s CUDA, which allowed programmers to ignore the underlying graphical concepts in favor of more common high-performance computing concepts. Newer, hardware vendor-independent offerings include Microsoft’s DirectCompute and Apple/Khronos Group’s OpenCL. This means that modern GPGPU pipelines can leverage the speed of a GPU without requiring full and explicit conversion of the data to a graphical form.
CHAPTER 1. INTRODUCTION

1.2 Motivation

To improve the signal processing performance, parallelization can occur both to data and tasks. With data parallelization the required computational time can be reduced by applying a single operation to multiple datasets simultaneously. With task parallelization, multiple threads can operate to the same dataset asynchronously and increase the utilization.

Even though data parallelization is supported by modern CPUs, the GPUs are preferred due to their high number of cores. Since GPP became commonplace, frameworks like NVIDIA CUDA [6] emerged. Apart from simplifying the GPU programming, CUDA also provides asynchronous queues called streams. Operations within the same stream are executed in issue-order. As a result, if an operation is blocked, all consecutive operations on the same stream will be blocked and the GPU will remain idle. This can be avoided by having multiple asynchronous tasks in flight to saturate the GPU (Fig. 1.4).

Existing CPU streaming engines efficiently handle the task parallelism, but they miss GPU data parallelism. There have been efforts to embed GPU support to these engines, but they are limited and constrained by the core’s original design.

![Image of Serial and Parallel Execution of Parallel routines]

Figure 1.4: Moving from sequential to parallel solutions

1.3 Related work

Demands on data analysis of big data have contributed to the development of: message passing frameworks like Kafka [7], batch streaming engines like Flink [8] and realtime computation system like Storm [9]. Because these systems are designed for stage-wise processing, it is not strange that they share common features with signal processing engines like Redhawk [10], Ossie [11], WarpLab [12], Matlab [13] and GNU Radio [14]. The latter
has been the favorite tool of SDR community for many years and multiple extensions [15] [16] have been built for it.

Works like [17] [18] [19] [20] [21] [22] have tried to offload the CPU by using GPU for the computational tasks, but only [23] tried to integrate GPU as part of GNU Radio’s core.

1.4 Calypso Contribution

Calypso is a batch streaming engine composed by a) CPU orchestrator emphasizing on task parallelism and b) GPU workers emphasizing on data parallelism. GPU workers are abstracted as asynchronous, self-contained CPU entities called modules. The modules can be placed in order by the end-user to form signal processing chains.

The engine is dedicated to minimize the end-user intervention by automatically handling the asynchronous operation, synchronization and pipelining of individual modules. Further, it takes care of operational duties like data exchange and memory management, so that the module-developer can be focused on the functionality part.

Calypso’s versatile and self-timed design makes it suitable not only for Real Time Computing with strict time-constrains but also for High Performance Computing where the throughput is the requisite.
Chapter 2

Calypso

Calypso can be regarded as a graph of inter connecting nodes, called "modules". They are self-contained entities associated to different signal processing functionality. Connections between nodes are one-way and represent the dataflow. Modules are constituted by a) an upper CPU layer written in C++11 handling the task management, synchronization and communication, b) a lower computational layer, written in NVIDIA CUDA that implements functionality.

The upper layer are to ensure the availability of input and output buffers. The input originates from the preceding module. For the output buffer, each module must issue a request to the CPU memory manager.

As the computations take place in GPU, the incoming and outgoing data are located in the GPU memory. To reduce the data transfer overhead, Calypso uses a messaging mechanism of buffer descriptors. Each descriptor is a CPU structure which describes a GPU buffer (Fig. 2.1). It contains information like: memory address, size and owner are needed.

Fig. 2.2 exhibits the above architecture. Calypso's components: CPU memory manager and upper layers of a module are located within the yellow panel. GPU components: GPU memory manager and lower layers of a module are located outside the panel and are marked as red. The arrow, representing the data flow, is also red.

To ensure the asynchronous operation of every component, they are executed on dedicated threads and are associated to different CUDA streams, so that modules can continue running uninterruptedly, regardless to the status of the others.

Calypso's engine, is build around these features:

2.1 Features

Traffic patterns Telecommunication standards may assume different traffic patterns. It could range from constantly incoming data on a fixed rate (stream) to large chunks of data at irregular intervals (burst). Further, the standard may assume constrains like time delivery or/and minimum data rate. Calypso features a versatile memory schema with a commence-signal mechanism that makes it adjustable to any scenario.

Extensible To simplify the development of new modules, Calypso provides elegant system of class inheritance, dispatchers and callback mechanism and memory management. Thus the developer can easily convert a standalone GPU function to a Calypso module.
If the end-user desires to build a new process chain (application), the only thing that is needed, is to define how the modules are connected. The rest as automatically handled by Calypso.

**Transactional** To ensure data integrity, user-provided function are wrapped into transactions automatically. In case of transaction failure, the transaction can be rescheduled arbitrary number of times. If a certain failure threshold is exceeded, the developer can choose whether to terminate the process chain or discard the dataset.
Asynchronous. To boost the performance, Calypso deploys cross-layer asynchronous operations. The upper (CPU) layer of a module is executed on dedicated CPU thread and the lower (GPU) layer on a different CUDA stream. Modules communicate using asynchronous message passing. Even if a module is busy at the time, the message is queued and can be fetched when the module becomes available.

Block to Stream. One of the main challenges in stream processing, is to break an infinite sequence of continuous bytes (streams) to finite sequences of bytes (block) so that a block-based device like the GPU, can use them (Fig. 2.3). Contrary to other engines, this operation is handled automatically and transparently to the developer.

![Figure 2.3: Calypso automatically transforms streams to blocks and vice versa](image)

### 2.2 Module

The upper layer (orchestration) of a module is constituted by a) the memory and module connectors b) the message converter and c) the transaction handler. The lower layer contains the execution environment and the corresponding GPU function.

The core of the module is the transaction handler. It is based on the two phase commit protocol. In the first phase (start_transaction routine) the handler saves the previous state and blocks until data become available to the input port. When this happens, the incoming buffer descriptor is unmarshalled and the extracted information is used by the handler to issue a request for an output buffer, to the CPU memory manager. The requested buffer size depends on the number of incoming items, their type and their consumption rate, as will be explain in the Signature section. This information is also used to estimate the number of expected outgoing items.

When the buffers are assured, the control is passed to the user-provided callback function (by default is called "work"). This function is a CPU wrapper for the CUDA kernel (Listing 2.2) and is used to keep the kernel state.

After the kernel execution, the second phase of the commit protocol takes control (end_transaction routine). The reported state information is cross-checked with the expected state. If they match, the transaction is marked as successful and the results can be forwarded to the next module. Simultaneously, the incoming buffer must be released (Fig. 2.5). If the reported and the expected states do not match, the transaction is regarded as failed and can be rescheduled arbitrary number of times. If the failure continues, the developer can decide whether to terminate the process chain or discard the current dataset.
CHAPTER 2. CALYPSO

Listing 2.1: Skeleton of execution environment

```
1 /**************************************/
2 ***** Module file (.h) *****
3 **************************************/
4 // Callback function. Triggered by Calypso
5 work() {
6     execution_environment(args);
7 }
8 
9 /**************************************/
10 ***** CUDA Implementation (.cu) *****
11 **************************************/
12 // CPU Wrapper
13 void execution_environment(args) {
14     cuda_kernel<<<grid, block>>>(args);
15 ...
16 }
17 // GPU function
18 __global__ cuda_kernel(args){}
```

Figure 2.4: Analytic sketch of a module


```
template<class T>
struct BufferDescriptor
{
    T* beginptr; // Buffer Begin
    T* ptr;      // Virtual Buffer Begin
    uint qid;    // Pool’s span id (linear)
    void* owner; // Buffer owner
    cudaStream_t cuStream; // Cuda Stream
    size_t used_items; // Items within buffer
}
```

Listing 2.2: Buffer descriptor fields

![Diagram](image-url)

Figure 2.5: Synchronization method between modules

### 2.3 Memory Manager

Module own dedicated buffer pools to save the outgoing data. For a module to continue uninterruptedly the corresponding pool must contain at least two buffers, so that if one buffer is forwarded, the second can be used to save the results of the next dataset.

To calculate the output buffer size, the number incoming elements, the type and consumption rate are needed. The type and consumption rate are constant, but the number of incoming items may change across iterations. Subsequently, the required output buffer size changes.

The first time that a module issues a "grab" request to acquire a buffer, an adequate buffer is allocated and is assigned to the module’s pool. The request buffer size is kept as reference for future requests. If a following request is for a bigger buffer, the new buffer is allocated and its size is used as the new reference size. Instead of de-allocating the buffers when they are released, they remain in the pool and serve future requests. If a request is for a buffer smaller than the existing, one of the existing buffers is divided into smaller
buffers (sub-buffers) and serve the request. This challenge at this point is to maximize the number of accommodated sub-buffers, so that more requests can be served simultaneously.

The first approach for the allocation algorithm, is to use the buffer that closer fits the requested size. However, this can cause low buffer occupancy due to GPU’s adjacent memory requirement. If the available memory is not contiguous, it cannot be used. For that, Calypso uses a Cyclical allocation algorithm [24], modified maximize the accommodated sub-buffers with regard to memory space continuity. The algorithm takes as input the current request size and the average request size of a time window. At first, it will try to find all the available contiguous buffers that can serve the requested size. Out of them, the buffer which can also serve the "average request size" will be preferred.

![Memory structure](image.png)

**Figure 2.6: Memory structure**

**Index** Buffers must be globally and uniquely identified. To achieve that, memory index is built as a tree structure with buffers located at the leaves, in depth=3. Each depth represents a different variable.

Depth=1 represents the owner module. To identify a module effectively in C++ "this" directive is used. It returns the starting memory address of a module instance, which is unique within the scope of a host. If multiple hosts are to be used simultaneously, the hostname must be prepended.

Depth=2 represents the buffer type. Since multiple buffers can have the same type and be used interchangeably, this level can be regarded as a buffer pool. A module may have more than one pool: one for the final results and one for the intermediate.

Depth=3 points to a unique buffers within the pool. To do that, each buffer must have a unique pool-scope linear id.
**Locking**  With a binary lock ( mutex ) only one consumer and one producer can exist without risking data integrity. If there are multiple consumers, when the first consumer unlocks a buffer the second consumer may face data corruption.

This dictates the use of references instead of mutexes. Buffers are locked as many times as the number of the consumers. When a consumer unlocks the buffer, the reference number is decreased by one. A buffer is regarded as available, only when the number of references are zero.

Due to the memory allocation algorithm, a buffer may be divided into smaller sub-buffers. To simplify the locking it is preferable to lock the corresponding buffer-descriptors 2.6 than directly the buffer.

**Commence Signal**  Calypso can operate both per element and per batch, with the batch size depending on the problem’s nature. The first module of the process chain, called feeder, dictates the batch size and consecutive modules adapt their batch sizes accordingly ( will be discussed in detail in Signature section ).

In high performance computing the batches are bigger in order to minimize the transfer and invocation overhead. On the other hand, in real time computing where the delivery time is the requisite, the batches are smaller. However, to improve the performance, the feeder has a small waiting window in which it can accumulate tasks without compromising the time constrains.

**Backpressure**  Backpressure [25] refers to the situation of a system that receives data at a higher rate than the process rate. Instead of dropping data, the "pressure" ( or load ) is propagated back to the root of the process chain.

As in Flink [8], each module owns a cyclically used buffer pool. A module can continue serving requests as long as there are available output buffers in the pool. When the buffers are exhausted, the module will block until a buffer becomes available. More memory means that the system can buffer away certain transient backpressure. Less memory means more immediate responses to back pressure.

If a throttler is placed at the output, it will cause backpressure up to feeding module which will be forced to push data into the process chain in a lower rate.

**Items**  To decompose a problem, it is preferable to use problem-based primitives rather than bits and bytes. Calypso groups arbitrary number of bytes in an entity called "item". An item is a collection of bytes that can be used similarly to a data type. For instance, a item may be a float or a Transport stream (TS) packet with 188 bytes. Items are templated C++ classes which can accommodate access and format methods. For instance, a double item may be presented as double, as 2 floats or as a sequence of bits.
CHAPTER 2. CALYPSO

Listing 2.3: An item is a class defining a data structure

Figure 2.7: Modules are connected using consistent queues, and signatures indicate the consumption rate of each module

2.4 Signature

Mathematically a module is a data transformation function. In Calypso, where the basic process unit is the item, this is translated as transformation of one item to another. In the simplest case, one incoming item matches to one outgoing item, regardless of their type. Nevertheless it is possible for each consumed input item to produce more than one output items. For instance, a complex number results to two float items.

The manifestation of consumption to production rate is called signature and is essential to describe a module. In particular, the signature describes the number of items that will be consumed and the number of items that will be produced for a batch size=1. Bigger batches, are proportional to that number.

As in Fig. 2.7, the first module consumes one item to produce 8 items of type $T_{1-2}$. 

```c
typedef GP<188> TS_packet;
typedef GP<12, 126> SYMBOL_2k;
struct GP
{
    public:
        union
            {
                uint8_t _1d[N * M];
                uint8_t _2d[N][M];
            } data;
        int size
    { return sizeof(this); }
};
```
The second module consumes these 8 items as pairs and produce a single item of type $T_{2-3}$. The output type of a module must be the same as the input type of the receiving module and type of the connecting queue.

If the first module had produced 7 items instead of 8, it would not be possible, for the second module, to consume all of them as pairs. While other frameworks leave the handling of this case to the developer, Calypso handles it automatically and transparently.

### 2.4.1 Outbound Fractional

When a module has produced more items than the manifested it is said to be outbound fractional. For instance, a module that has produced 10 items while the signature manifests that the output will be in triplets. The 10 items can be grouped in 3 triplets and one item will remain unmatched. To guarantee the stream continuity, this item must be grouped with the first two items of the next incoming buffer, to form a triplet.

In memory section it was mentioned that a pool must contain at least two buffers. One of the reasons is that the remaining item is stored in the secondary buffer while the primary buffer is forwarded. At the next iteration, the secondary buffer becomes primary and starts saving data after the first item.

### 2.4.2 Inbound Fractional

When a module needs to consume the input in a format other the manifested, it is said to be inbound fractional. This is complementary to the outbound fractional. For example, the signature manifests that the input is in triplets, but for implementation reasons it is easier to consume the input as pairs.

To solve that, Calypso extends the notion of signatures to sub-signatures. Oppositely to signatures which have the constrain that the input type must be the same as the output type of the preceding module, sub-signatures have no constrains. Any type can be used.

In the simplest case, signatures are converted to sub-signatures using type cast (buffer descriptors must be changed accordingly). However, in realistic scenarios the conversion are not integral and remaining bytes must be handled properly. The proposed solutions are:

- **Least common multiple**  Incoming items must be proportional to the least common multiple of signature and sub-signature. For example, the signature states that incoming items are 4 bytes and sub-signatures dictates them to be used as pairs (3 bytes). If the incoming buffer contains multiplications of 12, they can be consumed as 4 items, of 3 bytes each. Although it is quite effective and simple, it may increase the memory footprint significantly.

- **Data merge**  This approach resembles the outbound fractional. Incoming buffer is consumed based on the sub-signature and the remnants are stored to a temporary buffer. On the next loop, the remnants are merged with the new incoming buffer.

  Each buffer has foreseen an empty space equal to an item, so that remaining half-items can be directly merged with the buffer. This must happen with regard to the number of items contained within the buffer. It is possible to merge two half-items and make a complete item, as in Fig. 2.8.
CHAPTER 2. CALYPSO

Output buffer request

When a module issues a "grab" request for an output buffer, the request contains the number of the foreseen outgoing items. At this point, the module is aware only of the items contained within the input buffer and the consumption rate manifested in the signature. To calculate the output buffer size it is necessary to calculate the number of outgoing items as in Listing 2.4.

The feeder dictates the number of concurrent items and consecutive modules adapt appropriately to that rate based on the their signatures. Accordingly to Memory section, if a request is for more items than the available buffers, a new buffer of sufficient size is be allocated. However Calypso features one more experimental mode, in the developer can choose to split the exceeding items in a second transaction.
2.5 Topologies

The possible ways of connecting modules is called "topology". The simplest topology is the linear, in which the modules are placed in a row. Despite its simplicity it lacks flexibility and is used only to a limited number of cases. If a module is slower than the others, it backpressures the load as explain in the "backpressure" paragraph.

A generalization of the sequential model, is the branch model. In this model, the results are pushed to more than one receiver modules ( one to N ). To realize that, the use of references is mandatory ( see locking section ).

The symmetric topology, is called muxing. The input is constituted of multiple incoming links instead of just one as in Fig. 2.9. The muxing module blocks until input becomes available at every incoming link.

![Figure 2.9: Graph topologies supported by Calypso](image)
2.5.1 Slow module scenario
If there are multiple branches, the developer must decide whether they should be consistent or not. If they are consistent they must perform step-locked processing. If they are not, the slower branch is tolerated to miss a few datasets.

This can be useful when a branch needs only a subset of the incoming data. For instance, a branch may need to process only K sub-channels of a channel with N sub-channels (where K < N). To an observer this seems as the second loses data, but what happens is that the branch is not interested on this dataset and skips it.

2.6 Deeper into components

2.6.1 Memory Functions
Memory manager is a standalone Calypso component access only by the base classes (process, endpoint) using the API of Listing 2.5. The allowed sequence of operations for a buffer is to be register to a buffer pool, be grabbed by the owner module and be released by the consecutive modules of the owning module.

```c
void register (void *callerid, T* buffer);
void grab (void *callerid, T *&buffer);
void release (void *callerid, T* buffer);
```

Listing 2.5: Memory Manager API

**Register** Although new buffers are automatically allocated and assigned to the pool on demand, it is possible for a module to allocate its buffers manually. The allocated buffer has to be wrapped in a buffer descriptor, and this buffer descriptor must be used as argument to the register function, along with the ownerid. As for the owner id, as mentioned, the C++ "this" keyword is used.

**Grab** Grab is responsible to return a buffer equal to the requested size. The allocation algorithm is explained in the beginning of the memory management section. For a buffer to be regarded as available, the corresponding number of references must be equal to zero.

If there is no available buffer and the developer has the automatic allocation disable, the manager will have to wait for a buffer to become available. Instead of sleeping and polling, conditional variables are used (see optimization chapter).

**Release** Complementary to the grab, is the release function. A module locks one of its owning buffers and when it is populated with results, it is forwarded to the listening modules. Each listening module is responsible to unlock the buffer when its done. When a listening module releases the buffer, what happens is that the reference count is atomically decreased by one. The buffer will remain locked as long as there are references to it.

2.6.2 Classes
If every module had to re-write the aforementioned techniques, the cost and time of development would be significantly higher. Moreover, the developer would have less to focus on functionality and it would be prone to bugs. To avoid it, Calypso’s modules come
as a set of base classes. The developer has only to provide the corresponding execution environment.

Depending on their relative positions the modules have different characteristics and must inherit different base classes. Modules located at the edge of the process line acts as I/O to the chain. They are responsible to communicate with external sources, which means that one of their link-side is located within the GPU and the other on the CPU.

On the contrary, process modules are located within the process chain and are executed solely in GPU (2.9).

**Endpoint nodes**

**Input** An input endpoint is the first module of the process chain and is responsible to feed data to the chain. Depending on the resources on the CPU edge, it can be categorized as a) disk-based b) memory-based c) network-based.

Its default behavior is to fetch data from external sources at each iteration. However this can be a bottleneck especially for network-based modules. Instead, it is possible to fetch a great deal of data and save them for future usage. Subsequent requests will be served from the pool instead of getting data from the external source. If the remaining data fall over a threshold, new data are fetched asynchronously.

**Output** Compared to the input endpoints which must have contact at least one external source, this is optional for the output endpoints. Specifically an endpoint could acts as a) printer, for printing to a screen or to file b) measurer, for measuring the process chain throughput c) terminator, for terminate an process chain either loop-based or time-based d) dumper, which is similar to a terminator without terminating the process chain. This is necessary because every module must forward somewhere the results.

**Process nodes**

Fig. 2.10 sketches the control as happens within a module, to provide the functionalities as described so far. Specifically, a module consists of a part written by the developer (upper part in figure) and a base class which implements the Calypso protocol. Each module, becomes part of Calypso by inheriting this base class.

Analytically the steps taken for a module, from initialization to data output are listed bellow. Initialization phase involves steps A and B while steps C to G are the execution loop of each iteration.

A When a Module is instantiated, Base class constructor is called. In this step the module registers itself and the corresponding links in Calypso.

B After the Base class instantiation, the control returns to module’s constructor and module-scope initialization begins.

C Transaction handler triggers the input routine which blocks until there are available data on the incoming links and the corresponding output buffer is locked.

D When I/O buffers are ensured, the callback function is called.

E Callback function executed the corresponding function wrapper, which on its turn executes the CUDA kernel.
Figure 2.10: Module Overview, with steps followed from initialization to broadcast. After G control return back to D and waits until next input.

F When the function has finished, it reports back to the transaction handlers which decides whether the transaction was successful, or failed and need to be re-executed. If failed, the process goes to step E.

G If transaction is successful, the handler triggers output routine which forwards the (result) buffer descriptor to the listening module and releases the input buffer.

Currently, only one dataset can be handled at a time from a module. It would be possible to have multiple concurrent datasets by spawning multiple threads but extra effort is needed to keep state consistent across instances. Additionally, there is the need of an extra routine to ensure that data are published in the proper order.

2.7 Interface

Reference
Calypso makes extensive use of C++11 R-values (see Appendix) to minimize the overhead of copying variables. When defining a module’s constructor, the developer may choose to pass arguments explicitly as references as seen in the snippet bellow. However, if the module has more two incoming or outgoing links it cannot be resolved automatically which is which.
Variadic templates (see the Appendix) are used to pass arbitrary number of arguments to a function as in the following snippet. The drawback is that similar to the references, it may not be possible to identify whether a link belongs to the input or to the output list.

```cpp
// Definition
template<typename ... T>
Endpoint( params, T &&...links );
```

Listing 2.7: Variadic-based interface

**Tuples**

To solve the previous categorization issues, the proposed solution uses C++11 tuples. They are objects that pack elements of different types together in a single object, similar to a plain struct. Oppositely to struct which is a runtime entity, tuples are compile-entities. This requires a highly advanced techniques to unpack them properly at compile time. More specifically, a secondary template is used as iteration to the template holding the arguments. More technical details can be found in the Appendix.

As the snippet shows, it is possible to group incoming links into a tuple and outgoing links into another tuples. This makes feasible to have two separate list of the links. Similarly someone could claim to pack the arguments as simple structs. Unfortunately this would cause compilation errors because, as said, structs are run-time objects and cannot be used in the constructors.
CHAPTER 2. CALYPSO

2.8 Gnuradio vs Calypso

The GNU Radio architecture [26] is made for developing real-time signal processing, but
the processing blocks are not aware of the time elapsed. They simply run as fast as they
can to process their input data stream and send the result to their output. For example,
even if the Sampling Rate parameter is assigned to a Signal Source block, the signal
source block will generate as many samples as the CPU will permit every second. This
parameter value is only used to compute the corresponding digital frequency depending
on the sampling rate and frequency requested. In order to limit the processing power or
to set a real sample rate, you use either a Throttle block or a hardware driver sink. The
Throttle block will allow only a certain number of samples to pass through it in a given
time, causing all blocks in the same data stream to run at the specified sample rate.

On the other hand, Calypso’s behavior is quite different. It blocks when there are no
input items instead of producing dummy samples. Hence, it keeps a low system footprint
when remain idle. However, it is possible to emulate the same behavior in Calypso by
using a signal generator at the input of the feeder module.

Calypso could be described as event-driven while GNU Radio as sample-driver engines.
Chapter 3

Optimisation

3.1 Algorithmic

3.1.1 Parallelization levels

To efficiently parallelize a problem apart from the mathematical approach, the developer must also take into account the available hardware [27]. Although a problem may be theoretically decomposed to multiple parallel tasks [28], it is not necessary that the hardware will support this level parallelization.

For instance, a Single Instruction, Multiple Data (SIMD) processor can apply the same operation to multiple data [29], but cannot execute multiple instances of a function to different datasets [30]. Additionally, in extremely parallelized cases the overhead of spawning multiple threads may overwhelm the benefits per thread.

Based on the coupling type [31], the developer can decompose the problem to parallelization units as follows:

- **Item-span**: describes extremely parallelized cases in which there is no coupling on the data. In bibliography this is known as embarrassingly parallel.

- **Item-wise**: describes cases that data within an item are strongly coupled and cannot be parallelized. However, items are loosely coupled and multiple items can be processed in parallel.

- **Pattern-wise**: extents the item-wise. Strongly coupled items, but loosely coupled patterns. For instance, pseudo-random sequences are repeated sequences every N items.

3.1.2 Lookup

Regularly in signal processing the same values must be calculated at each iteration. That should not be a problem when the computation is fast. However, when intensive computations (e.g. polynomial) are needed, the performance will be degraded. The use of lookup tables can greatly improve the performance by reducing the necessary calculation to access time. Required values are calculated at the initialization and are saved into a map. When needed, the application advice from this map the values.
3.2 Software Optimization

3.2.1 Memory Alignment

Due to the way CPU work, proper alignment at offsets of power of 2 can blast off the performance. Otherwise, CPU will waste cycles to handle the misalignment automatically. Given the modern high-capacity memories it is advisable to pad structures with dummy data to comply with the alignment constrain. Listings 3.1 and 3.2 illustrate an example of poorly ordered and properly ordered structure.

In the case of the poorly ordered structure the occupied space is

\[ 1(\text{bool}) + 7(\text{padding}) + 8(\text{double}) + 2(\text{short}) + 2(\text{padding}) + 4(\text{int}) = 24 \text{bits (3 bytes)} \]

while in the case of the properly ordered structure

\[ 8(\text{double}) + 4(\text{int}) + 2(\text{short}) + 1(\text{bool}) + 1(\text{padding}) = 16 \text{bits (2 bytes)} \]

Listing 3.1: Poorly ordered

```c
// Total 24 bits (3 bytes)
struct {
  bool b;
  double d;
  short s;
  int i;
};
```

Listing 3.2: Properly ordered

```c
// Total 16 bits (2 bytes)
struct {
  double d;
  int i;
  short s;
  bool b;
};
```

3.2.2 Cache Coherence

When a program access a chunks of data, it is highly probable that its neighboring data will be access soon. This phenomenon is called data coherency. The developer can gain from the fact that CPU automatically prefetches neighboring data, by choosing the proper dimensions. The problem is illustrated at Listings 3.3 and 3.4.

While in the coherency case the data are access sequentially, in non coherency case data have a distance of 32 bytes making CPU prefetching useless.

Listing 3.3: Coherent Data

```c
for (int i=0; i < 32; i++)
  for (int j=0; j < 32; j++)
    total += myArray[i][j];
```

Listing 3.4: Non Coherent Data

```c
for (int i=0; i < 32; i++)
  for (int j=0; j < 32; j++)
    total += myArray[j][i];
```

3.2.3 Structs and Arrays

Closely related to the data coherency, is the problem of data within a structure. Assuming N elements of type A, B, C, the developer must decide whether to treat them as array of structures (Listing 3.5) or as structure of arrays (Listing 3.6). Whenever the are to be used
as a record, it is preferred the use of array of structures, because multiple records can be served simultaneously. On the other hand, if there are frequent vectorized operations - same operation on multiple data - the use of structure of arrays is preferable for performance issues.

However, the memory alignment must be taken in mind. Since the compiler automatically does the padding, it may result to large memory footprint for Array of Structures (AOS) because each record is padded separately. For Structure of Arrays (SOA) the padding is only at the end of the array (Listing 3.7), reducing the required memory.

On a side note, in AOS records are accessible by multiplying the index by the struct size. For struct sizes in power of two the multiplication can be replaced by bit shifting, which greatly improves the overall performance.

```
struct {
   double x;
   int y;
   char z;
} AOS [N];
```

Listing 3.5: Array of Structure

```
struct {
   double x[N];
   int z[N];
   char y[N];
} SoA;
```

Listing 3.6: Structure of Arrays

```
// AOS
double | int | char | *pad* | ... 98 structs... double | int | char | *pad

// SOA
| double | ... | double | *pad* | int | ... | int | *pad* | char | ... | char |
```

Listing 3.7: Memory footprint of AOS and SOA

### 3.3 Effective coding

#### Power of two

Division is optimal when the divisor is power two. The arithmetic operations are emulated by bit shifting. Similarly, when applicable is preferred to use bitmask operations than arithmetic operations. Especially for modulo which is frequent in signal processing, it is advisable to replace \( x = a \mod 4 \) with \( x = a \& 0x03 \) to increase the performance.

#### Bit stretching

To simplify the bit operations, a method called bit stretching is used. Bytes with 8 useful bits, are replaced by 8 bytes of 1 useful bit. Although it increases the memory footprint, it makes the operations easier for the developer and in some architectures it may also increase the performance.
CHAPTER 3. OPTIMISATION

Blocking

In concurrent programming frequently an operation must block to wait an event. Common pitfalls are either to go into sleep for long periods or continuously poll for the event, which results to high system load.

Conditional variables are synchronization primitives used to signal events across threads. Each conditional variable is associated to a mutex, which is a associated to a resource. Listings 3.8 and 3.9 illustrates the usage of conditional variables for a consumer and a producer for a concurrent queue, respectively.

```
// Listing 3.8: Consumer snippet

std::unique_lock < std::mutex > mlock( mutex_ );
...
cond.wait( mlock );
...
queue_.pop();
```

```
// Listing 3.9: Producer snippet

void push(T item )
{
    std::unique_lock < std::mutex > mlock( mutex_ );
    ...
    item.push( ...
    mlock.unlock();
    cond.notify_one();
}
```

3.4 Hardware optimization

3.4.1 Scalar vs Vector processors

Originally, CPUs were designed to execution sequential code, instruction after instruction, dataset after dataset. This type of processors are called scalars. A scalar is an atomic quantity that can hold only one value at a time. In Flynn’s taxonomy these processors are categorized as Single Instruction, Single Data (SISD). Despite their simplicity, they suffer low performance. For instance, to multiple a buffer of 100 integers with a number, it would take 100 iterations (one per item), or else O(N) complexity.

Contrary to the scalar processors, which apply a single instruction to a single dataset, Single Instruction, Multiple Data (SIMD) apply a single instruction on multiple datasets (Figure 3.1).

3.5 NVIDIA CUDA

While SIMD boosts arithmetic operations, they cannot handle logical entities. To go beyond this, NVIDIA has introduced a new model called Single Instruction, Multiple Threads (SIMT) (see the Appendix). Instead of applying a single instruction to a dataset, Single Instruction, Multiple Data (SIMD) apply a single instruction on multiple datasets (Figure 3.1).

Even further, NVIDIA has developed Compute Unified Device Architecture (CUDA) which hides complex GPU primitive, like vertices and shaders, to a C++ like language. CUDA is also capable of asynchronous operations, called streams. Streams are sequences of operations in issue-order on the GPU. Each GPU operations is associated to a stream. For operation within the same stream, issue-order is guaranteed. This however, implies no order guarantee for operation in different streams. If no stream is specified, the operation inherits the default zero stream. Compares to other streams which provide asynchronous execution with regard to the host, the zero stream is synchronous; CPU stalls until the GPU execution has finished.
3.5.1 Hardware Arrangement

NVIDIA’s Graphics Processing Unit is constituted by multiple Streaming Multiprocessors (SM) where each is further constituted by multiple Streaming Processors (SP).

In each SM only one operation can be active at a time, with all the SP applying this operation to the data. If the number of scheduled threads exceed the number of physically available SPs, they are scheduled for execution. To optimize the dispatching, every 32 threads make a group called warp [32] and warps are distributed to SP.

If a warp is blocked, another scheduled warp replaces it. Therefore idle times are minimized, utilization is increase and with it there is an increased throughput.

3.5.2 indexing

CUDA abstracts task distribution to SMs and SPs as dimensions. Specifically:

**Thread** represents the execution of a function instance. Each thread is executed in a separate SP of a SM. As Figure 3.3 shows, a thread has access to registers, shared memory of the SM and global-scope memory.

**Block** Threads are organized in blocks (Figure 3.2 and 3.3). A block is assigned to a SM. The threads of a block can be accessed using 1D, 2D or 3D indices. Blocks cannot
be split and be distributed to multiple SM. Therefore, threads of a block can access only the shared memory of the SM handling the block. Threads can be coordinated using _syncthreads() function.

**Grid** Threads compose a block, and multiple blocks compose a grid. Each thread is assigned to a SP and each block to a SM. Similarly, a grid is assigned to a single GPU chip. This is useful for GPUs with multiple chips, or for multiple co-existing GPUs.

The warp size is the number of threads running concurrently on an MP. In actuality, the threads are running both in parallel and pipelined. At the time this was written, each MP contains eight SPs and the fastest instruction takes four cycles. Therefore, each SP can have four instructions in its pipeline for a total of $8 \times 4 = 32$ instructions being executed concurrently. Within a warp, the threads all have sequential indices so there is a warp with indices 0..31, the next with indices 32..63 and so on up to the total number of threads in a block [33]. If an warp is not ready or is blocking, it will be immediately replaced by another in order to minimize the core’s idle time.

### 3.5.3 Memory Hierarchy

An in most distributed systems, CUDA trades memory access time with memory capacity. From higher capacity to lower:

- **Global memory**: This memory is built from a bank of SDRAM chips connected
CHAPTER 3. OPTIMISATION

Figure 3.3: Cuda memory hierarchy. Each memory has a trade-off of memory speed vs memory size

to the GPU chip. Any thread in any MP can read or write to any location in the global memory. Sometimes this is called device memory.

- **Texture cache**: This is a memory within each MP that can be filled with data from the global memory so it acts like a cache. Threads running in the MP are restricted to read-only access of this memory.

- **Constant cache**: This is a read-only memory within each MP.

- **Constant cache**: This is a small memory within each MP that can be read/written by any thread in a block assigned to that MP.

- **Constant cache**: Each MP has a number of registers that are shared between its SPs.

3.5.4 Warp divergence

Because all the SP of an SM execute the same operation on their data, it is said that they are in lock-step execution. Therefore, if a thread of a warp falls over different execution path (e.g. for one thread the conditional is evaluated to true, while on the rest is false) all the threads must stop the execution until the different execution branch is finished and they converge again. However, during this time is possible that other wraps replace the stalling warps, but they cannot proceed after stalling point.

It may not be always possible to avoid conditional branching, but frequently there are tricks that can result to the same behavior. Listing 3.10 illustrates an example of how is possible to move decision making from GPU to the CPU. This in in accordance with NVIDIA’s current trend to replace decision circuits with their smaller arithmetic counterparts.
template<int action>
__global__ void kernel()
{
    switch(action) {
    case 1:
        // First code
        break;
    case 2:
        // Second code
        break;
    }
}

template void kernel<1>();
template void kernel<2>();

Listing 3.10: Templated Branch unroll
Chapter 4

DVB-T

Digital Video Broadcasting Terrestrial (DVB-T) [34] is the European-based consortium standard for the broadcast transmission of digital terrestrial television. This system transmits compressed digital audio and digital video as an MPEG transport stream, using coded orthogonal frequency-division multiplexing modulation.

It is used as evaluation testbed to Calypso and GNU Radio. It is selected because it requires both block-based routines and stream-based routines with multiple incoming/outgoing byte rate. Although the implementation is rather challenging, its components are used in multiple Telecommunication standards. Therefore, the evaluation results can be used as make general conclusions. And above all, it is fancy since it can broadcast to any digital TV.

4.1 Integration to Calypso

To promote the code re-usability, each file functionality is implemented in a different file. This also helps to separate the compilation of CUDA kernel with nvcc and the Calypso with gcc (with c++11 flags). Listing 4.1 illustrates the linking of independent CUDA kernels and Calypso.

When the module's constructor is called, before anything else, the Base class constructed is invoked. Although the module constructor depends on the developer, for the Base class constructor is necessary to include signature information like a) number of items that makes an input unit b) number of items that makes an output unit c) incoming links d) outgoing links.

Next, module initialization take place and the control returns to the caller. Then, the caller uses Calypso to pass the control to the callback function. The "work" body is wrapped into a transaction with error handling routines. The function blocks until input and output buffer become available. When they are ready, the control is passed to the CPU function accompanying the CUDA kernel. To properly use a kernel, arguments must contain at least a) input buffer pointer b) output buffer pointer c) number of items d) CUDA stream.
CHAPTER 4. DVB-T

32

Listing 4.1: Module definition, and wrapper function call (execution environment)

Listing 4.2: Wrapper function definition, and CUDA kernel call

4.2 Scrambler

The scrambler uses as data source a TS, which is a standard container format for transmission and storage of audio, video. It is composed by a) a sync byte b) a header c) optional transport fields d) and the actual payload. Packets are 188 bytes in length.

Its purpose is to encode a message at the transmitter so that it is unintelligible if the receiver is not equipped with a descrambling device. It differs from encryption in the sense that encryption is carried out in the digital domain whereas the scrambling in the analog domain.

It is also useful for accurate timing recovery on the receiver without resorting to re-
dundant line coding. Further it is used to de-correlate the signal’s power spectrum from the actual transmitted data by breaking long sequences of ‘0’ or ‘1’.

In DVB-T standard the applied transformation Pseudorandom binary sequence (prbs) is depicted in Fig. 4.1 \( S(x) = 1 + x^{14} + x^{15} \).

To avoid the overhead of calculating the polynomial at each loop, it is calculated a-priori and is used as a lookup table. On top of that, the prbs is a finite sequence repeated every 8 TS packets. This fact can be exploited to launch tasks on multiple, independent sequences (pattern-wise parallelization).

Nevertheless, the developer must take into account the state keeping across tasks. If the input is 20 TS packets, it can be decomposed to two groups of 8 TS packets and one group of 4 TS packets. At the next loop, the operation must start with an offset of 4 packets.

![Scrambler Schematic diagram](image)

**Figure 4.1: Scrambler Schematic diagram**

### 4.3 External Encoder

Data transfers over wireless networks or even within a host bus are prone to errors. To increase the robustness, additional error correction information is appended to each transferred packet.

DVB-T uses a Reed-Solomon RS(204, 188) [35] code which enables the correction of up to a maximum of 8 wrong bytes for each 188-byte packet. The shortened RS(204, 188) code may be implemented by adding 51 bytes, all set to zero, before the information bytes at the input of an RS (255,239, \( t = 8 \)) encoder. After the RS encoding these null bytes are discarded, leaving an codeword of \( N = 204 \) bytes.

For performance reasons, arithmetic operations like multiplication and division can be replaced by their logarithmic counterparts. For instance

\[
10001001 \ast 00101010 = \alpha^{74} \ast \alpha^{142} = \alpha^{74+142} = \alpha^{216} = 11000011
\]

Here, the problem has been reduced to finding the power of \( \alpha \) that corresponds to the value 10001001. Using the properties of Galois fields, one can easily construct a lookup table with log, antilog (exponential) for every possible value.

Contrary to the scrambler which can be highly parallelized within the span of an item, it is almost impossible to do that with the Reed-Solomon due to the strong data dependency.
However, different TS packets are completely independent and be used as parallelization units (item-wise).

4.4 External Interleaver

Error correction codes can fix only a limited number of flipped bits per item. If the error occur as burst (long sequences of errors) the correction will not be feasible. To prevent it, data re-shuffled in a deterministic way. Even if an error-burst occur to the re-shuffled data, when re-assembled, the error will be spread on the original sequence.

Convolutional interleaving (Fig. 4.2) is a byte-wise interleaver that permutes bytes to a predicted offset. It is constituted of 1 = 12 branches, cyclically connected to the input byte-stream as shown in Fig. 4.2. Each branch j shall be a First-In, First-Out (FIFO) shift register, with depth \( j \times M \) cells where \( M = 17 = N/I, N = 204 \).

![Figure 4.2: DVB-T convolutional interleaver outline](image)

Similar to the scrambler’s prbs, the offset for each byte is calculated a prior. However, one must take into consideration that a the last branch 12 has a maximum delay equal to depth = 17 \( \times \) 11 = 187. Elements are queued to branch 12 every 12 bytes resulting a maximum offset = 12 \( \times \) depth = 2244. Normalized this is packet_offset = 2244/204 = 11, which dictates the history keeping of the last 11 packets.

4.5 Internal Encoder

The previously used ReedSolomon encoder is a hard-decision algorithm and requires large block length to minimize the false decoding probability. However larger blocks imply higher decoding complexity.

In 1966 Dave Forney conceived the convolutional encoders which allow exponentially decreasing error probability with increasing block length and polynomial-time decoding complexity.
Typically, the inner code is a soft-decision convolutional Viterbi-decoded code with a short constraint length. It is used in collaboration with Reed-Solomon to increase the overall error correction capabilities of the system. Convolutional codes generate parity symbols using a polynomial function and incoming data.

Normally, this type of encoders rely on bit-wise operations, but for programming efficiency a technique named "byte stretch" (Listing 4.3) was deployed. A byte of 8 useful bits is stretched into 8 bytes of 1 useful bit each. Since a convolutional encoder produces two bytes for each consumed byte, using the "byte stretching", each byte results to 16 bytes (2 bytes from the encoder, 8 bytes for each generated byte).

Unfortunately, none of the previous techniques can be used directly to optimize the procedure due to the streaming nature and data dependencies of the problem.

Using Calypso, the developer can overcome the problem by sliding the encoder over the data, instead the data over the encoder. To enlighten the difference, assume the case of a simple encoder which operates on the first and the third element. Encoders can be applied to pairs [0,2], [1,3], [2,4] simultaneously.

![Figure 4.3: The mother convolutional code of rate 1/2](image)

**Listing 4.3: Byte stretching**

```c
stretched_input[7] = (input[blockID] & 1) >> 0;
stretched_input[6] = (input[blockID] & 2) >> 1;
stretched_input[5] = (input[blockID] & 4) >> 2;
stretched_input[4] = (input[blockID] & 8) >> 3;
stretched_input[3] = (input[blockID] & 16) >> 4;
stretched_input[2] = (input[blockID] & 32) >> 5;
stretched_input[1] = (input[blockID] & 64) >> 6;
stretched_input[0] = (input[blockID] & 128) >> 7;
```

### 4.6 Puncturer

The information rate of an error correction code is the proportional of the useful data to the total data (useful + redundant). In noisy environments, redundancy is desired but in noiseless environments, it is waste of resources. To increase the code rate (increase goodput over throughput), parity bits are eliminated. This has the same effect as if the encoding
had happened in a higher rate (less redundancy). For simplicity reasons, it is preferred to add an extra component to throw away arbitrary number of bits than having a dedicated encoder for each case.

Compared to the previous with produce one item for each consumed item, the puncturer produces 136 items for each consumed item. The reason is that it is preferred to have the producer module do the data management than assigning it to the consumer module. The producer has the advantage of knowing the exact number of items within the buffer and if exceed the contract constrain, they can be moved to the secondary buffer. On the contrary, if data management was happening at the input stage of the receiving module, complex time-based or size-based mechanism would be needed. Therefore, the puncturer is responsible to forward data in a format convenient for the receiving module.

Specifically, as in Fig. 4.4, the interleaver expects items of size 6048 bytes and puncturer produces items of 24 bytes. To compromise, the puncturer groups 252 items of 24 bytes to make an item of 6048 bytes.

If the developer had to do the grouping, it would break both code-reusability and developer’s nerves. Rather, Calypso uses an extended signature. Additionally to ingress and egress descriptors there is one descriptor for intermediate consumption. It is used a transformation layer between implementation units and logical units.

As in the signature, an item of 3264 is broken into 102 items of 32 bytes each which are punctured and 102 items of 24 bytes are produced. Later, the corresponding intermediate signature of inner interleaver merges these 252 items into an item of 6048 bytes.

\[\begin{array}{cccccc}
X_0 & X_1 & X_2 & X_3 & X_4 & X_5 \\
Y_0 & Y_1 & Y_2 & Y_3 & Y_4 & Y_5 \\
\end{array}\]

\[\begin{array}{cccc}
X_0 & Y_0 & Y_1 & X_2 & Y_2 & Y_3 & X_4 & Y_4 & Y_5 \\
\end{array}\]

Figure 4.4: Outline of Puncturing R=2/3

### 4.7 Internal Interleaver

Similar to the outer interleaver, the encoder output is reshuffled again. It combines the interleaving with pre-modulation grouping. Specifically, the input is divided into \(\nu\) sub-streams, where \(\nu\) depends on the modulation schema; \(\nu = 2\) for QPSK, \(\nu = 4\) for 16-QAM, and \(\nu = 6\) for 64-QAM. The demultiplexing uses the mapping:

- \(x_0\) maps to \(b_{0,0}\), \(x_1\) maps to \(b_{2,0}\)
- \(x_2\) maps to \(b_{1,0}\), \(x_3\) maps to \(b_{3,0}\)

\[\begin{array}{cccc}
x_0 & x_1 & x_2 & x_3 \\
\end{array}\]

\[\begin{array}{cccc}
b_{0,0} & b_{0,1} & b_{2,0} & b_{2,1} \\
\end{array}\]
where the input bit $x_{di}$ maps to an $b_{e,do}$ using the formula:

$$x_{di} = b[di \mod v](\text{div})(v/2) + 2[di \mod (v/2)] \div (v/2)$$

$x_{di}$ is the input to the demultiplexer in non-hierarchical mode.

$b_{e,do}$ is the output from the demultiplexer.

$di$ is the input bit number.

$\text{mod}$ is the integer modulo operation.

$v$ is the number of substreams.

$\text{div}$ is the integer division operation.

Then, these sub-streams are mapped to a number of active carriers per OFDM symbol; 1512 for 2K mode, 6048 for 8K mode. Thus in the 2K mode, 12 groups of 126 data words are read sequentially into a vector $Y' = (y'_0, y'_1, y'_2, \ldots, y'_{1512})$, where

- $y_{H(q)} = y_q$ for even symbols for $q=0, 1, 2, \ldots, N_{\text{max}} - 1$.
- $y_q = y'_{H(q)}$ for odd symbols for $q=0, 1, 2, \ldots, N_{\text{max}} - 1$.

As had happened with the outer interleaver, the offset of a bit between the original and the permuted sequence can be predicted and can be parallelized easily.
4.7.1 Modulation

Modulation is the process of varying one or more properties of a periodic waveform, called the carrier signal, with a modulating signal that typically contains information to be transmitted. Another closely related definition is that modulation is the process of conveying a baseband signal (e.g., video signal) inside another signal that can be physically transmitted (passband). For the modulation DVB-T uses Orthogonal Frequency Division Multiplexing (OFDM).

The transmitted signal is organized in frames. Each frame has a duration of $T_F$, and consists of 68 OFDM symbols. Four frames constitute one super-frame. Each symbol is constituted by a set of $K = 6817$ carriers in the 8K mode and $K = 1705$ carriers in the 2K mode and transmitted with a duration $T_S$. It is composed of two parts: a useful part with duration $T_U$ and a guard interval with a duration $\delta$. The guard interval consists in a cyclic continuation of the useful part, $T_U$, and is inserted before it.

In addition to the transmitted data, each frame also includes scattered pilots carriers, continual pilots carriers and Transmission Parameters Signalling (TPS) carriers. They are used for frame, frequency and time synchronization.

**Constellation mapping** is the process of mapping incoming bytes into constellation points. As discussed in inner interleaver, data have been transformed so that each byte includes 4 useful bits (4-bit words). This is specific to QAM-16 which have been chosen. If QAM-64 were used, the constellation unit would be 8-bit words (1 byte).

To map these words into In-phase, Quadrature (IQ) plane the used transformations is:

\[
\text{out}[i].x = \text{constant\_qam16\_re}[\text{in}[i]]; \\
\text{out}[i].y = \text{constant\_qam16\_im}[\text{in}[i]]; \\
\]

where constant\_qam16\_re is a lookup table with values 3,3,1,3,1,1,-3,-3,-1,-1,-3,-3,-1,-1 and constant\_qam16\_im is a lookup table with values 3,1,3,1,-3,-1,-3,-1,3,1,3,1,-3,-1.

The result of the mapping is a populated IQ plane as in Fig. 4.6.

![Figure 4.6: Constellation points for QAM16](image-url)
### Pilot and TPS signals

Various cells within the OFDM frame are modulated with reference information whose transmitted value is known to the receiver. The information transmitted in these cells are scattered or continual pilot cells, and are used for synchronization issues.

On top, TPS information is needed to described the signal in terms of hierarchy, code rate and transmission mode.

The TPS is defined over 68 consecutive OFDM symbols, referred to as one OFDM frame. Each OFDM symbol conveys one TPS bit. Every TPS carrier in the same symbol conveys the same differentially encoded information bit.

The continual and scattered pilots are modulated according to a PRBS sequence. As had happened with the scrambler, to avoid the polynomial calculation at each step, it is calculated as the initialization phase and then the results are used as a lookup table. Similarly, the TPS positions are calculated once as is used as lookup.

Because TPS signal is spread within frame’s span while pilots while symbol’s span, it is preferred to spawn thread symbol wise than byte wise. This also helps to increase the effectiveness of a thread. If thread spawning is too fine-grained (e.g. increase the value by 1 and terminate) the creation overhead cost becomes significantly higher than the effectiveness of the thread.

### Inverse FFT

In transmitters using OFDM as a multicarrier modulation technology, the OFDM symbol is constructed in the frequency domain by mapping the input bits on the IQ plane and order them in a sequence with specific length according to the number of subcarriers in the OFDM symbol. That is by the mapping and ordering process, one constructs the frequency components of the OFDM symbol. To transmit them, the signal must be represented in time domain. This is accomplished by the inverse fast Fourier transform IFFT. Instead of reinventing the wheel, the cuFFT library provided by NVIDIA is used, with regard to the asynchronous streams.

### Guard Interval

To decrease receiver complexity, every OFDM block is extended, copying in front of it its own end (cyclic prefix). Cyclic prefix is required to operate single frequency networks, where there may exist an ineliminable interference coming from several sites transmitting the same program on the same carrier frequency.

### 4.8 Implementation real-time throughput

Table 4.1 presents the signature per Calypso module. The signature can be used to estimate the amount of output data for a module (the output is proportional to the
input), which later can be used to estimate the actual throughput of a module.

For the scrambler and the interleavers there is no data volume change because they only "reshuffled" the existing data. Encoders generate new data (either symmetric as in ReedSolomon, or asymmetric as in Convolutional encoder). On the other hand, puncturer decreases data volume by 1/3 (shrinking 32 bytes to 24 bytes).

Although it is said that interleavers do not change data volume, one may notice that in inner interleaver data are shrink by 1/4. This is implementation specific, and is happening because 4 individual bytes with 1 useful bit are merged into 1 byte with 4 useful bits.

Except for the data volume, the signatures are also very helpful to explain throughput changes. Obviously, if two modules generate different amount of data their throughput will be different. But if they generate the same amount of data, one can make conclusions for the module's performance based on the throughput.

To measure the impact of process chain compared to the individual performance, the stand-alone module must start with the equivalent number of items of the process chain. Or else, in form of question "How many items must the stand-alone Puncturer to emulate the behavior of process chain starting with 20 parallel items?".
Chapter 5

Evaluation

The evaluation of Calypso consists of behavioral and comparison experiments. Behavioral are focused on enlightening the correlation between parameters and output measurements. Later these parameters can be used to achieve a desired behavior (e.g. low CPU usage, high throughput, time constraints). The comparison experiments are used to compare the performance (throughput, CPU usage, total time) of different implementations. As a testbed, the DVB-T standard has been implemented in GPU as a Calypso application and in CPU as a GNU Radio application.

The skeleton of the experiments is a TS file (600 MB, 1 minute and 17 seconds) loaded in a ramdisk which is fed into the DVB-T process chains. The chain’s output is redirected to a named pipe at the end of which the mbuffer [36] is attached. Depending on the the case, mbuffer may act either as throughput measurer or/and throttler. Calypso’s and GNU Radio’s is measured by perf [37].

The experiments were conducted in an out-of-shelf workstation equipped with two Intel Xeon E5-2630 V2 @2.6GHz CPU, one NVIDIA Quadro K2000 GPU [38] and 32 GB DDR3 @1866 MHz. The operating system in use was a Fedora 20 with CUDA version 6.5 and GNU Radio version 3.7. As for the DVB-T, the selected standard was 2k mode with QAM16 modulation and 2/3 puncture rate.

5.1 Buffer size - Throughput

This behavioral experiment, tries to find the relation between the batch size and the output throughput. At each iteration the batch size is increased and the throughput is measured.

As can be seen in Fig. 5.1, less items (210) causes higher CPU utilization (blue line), lower throughput (bars) and higher process time (red line). This occurs because multiple small buffers require more CPU coordination, while the GPU remains idle for long periods.

As the batch size increases, the scenery changes. Throughput gets higher while CPU utilization and process time decreases. Bigger buffers require less memory transfers and can keep the GPU busy for longer periods. Nevertheless, at a point the GPU is saturated and can no longer serve requests. Therefore, it makes no differences if bigger buffers are used. This can be seen at the last 3 bars.

This experiment give guidelines for the next experiments. In particular, it shows that 2100 item result to the highest throughput. For the next experiments, the value 2100 is chosen in order to increase the output throughput.
CHAPTER 5. EVALUATION

5.2 Real time transmission throughput

For Calypso to be effective, it must be able to achieve at least the minimum throughput for a successful real time transmission.

At this point it would be helpful to make clear the difference between the theoretical throughput and the implementation throughput. The final throughput must be the same in both cases, but the intermediate throughput (for intermediate modules) may vary significantly.

In particular, for the Convolutional Encoder module the DVB-T standard defines bitwise operations. To improve the programmability, a bit-stretching method is used. Instead of having bytes with 8 useful bits accessed by bit-wise operations, each bit is assigned to a different byte (with 1 useful bit) and are accessed byte-wise. Later, these stretched bits are merged to bytes of 4 useful bits.

This is a loud example of how theoretical throughput may vary from the implementation throughput. Under the specific implementation, Convolutional encoder and puncturer require 8 times (1 byte is stretched to 8 bytes) the throughput of the minimum theoretical defined in the standard.

For this experiment, the output throughput of Calypso(GPU) and GNU Radio(CPU) is throttled to the theoretical throughput and at each iteration the performance of each module is measured. The scope of this experiment is to measure the required, per-module throughput to achieve a real time transmission.

As Fig. 5.2 shows, both Calypso and GNU Radio can achieve the minimum required throughput for a real-time transmission.

As explained in the DVB-T section, for the first three modules the incoming byte rate is almost equal to the outgoing byte rate. At the convolutional encoder, the outgoing byte rate should be 16 more than the input rate. Similarly, the outgoing byte rate of the puncturer should be almost one fourth of the incoming byte rate, and the same should happen for Inner Interleaver.
Figure 5.2: Per module throughput of Calypso(GPU) and GNU Radio(CPU) when output throughput is throttled to the minimum rate of DVB-T transmission

5.3 Maximize transmission throughput

The same methodology applies in this experiment, but without any throughput throttler. Both engines are left to perform their best.

Fig. 5.3 shows that Calypso(GPU) outperforms GNU Radio(CPU) in every module. One can see that the variation is the same while the absolute throughput is different. In both cases, backpressure methods are used and the preceding modules adapt their rates to the process rate of the last module. However, the last module in Calypso(GPU) performs twice as better than GNU Radio(CPU), and subsequently this stands for all the previous modules.

5.3.1 Required resources

The next question, is what are the consumed resources of Calypso(GPU) and Radio(CPU). To answer it, another set of experiment is needed. This time, experiments start with only the scrambler in the process chain, and at each iteration a new process module is added until the full DVB-T process chain is complete. At each iteration, the produced data, the total time and the total CPU of the last module are measured (Fig. 5.4).

As has been said, Calypso(GPU) assigns the computational part on the GPU while
CHAPTER 5. EVALUATION

Figure 5.3: Per module throughput of Calypso(GPU) and GNU Radio(CPU) when output throughput is not throttled

Fig. 5.4a shows that the less time needed spent in process, the more data coordination is needed. Indeed, buffers in lightweight modules like interleavers and puncturer finish fast and more orchestration in the CPU is needed. On the other hand, buffers in slow modules like encoders (ReedSolomon and Convolutional) and OFDM spend most of their time on the GPU and require less orchestration. Increased process time, also affects the total time required to finish an experiment. Fast modules contribute a negligible amount of time on the process chain, while slow modules can dramatically increase the total time.

A counter-intuitive fact of Fig.5.4a is that when the puncturer is the last module, the total process time is dropped instead of increasing. This can be explained as a combination of negligible process time and system-oriented reasons like scheduling algorithm, memory caching and memory coalesce.

In GNU Radio, as shown in Fig. 5.4b, the more modules are appended the more CPU is used. This happens because both orchestration and calculation take place on the CPU. Therefore, it is difficult to say which part corresponds to the orchestration and which to the calculation. However, Convolutional encoder has a very interesting behavior. Although it dramatically increases the total time, the CPU overhead is rather small. This leads to the conclusion that the encoder is blocked for a significant period, and it is the bottleneck

the GNU Radio(CPU) on the CPU. The orchestration, in both cases, is on CPU.
of the process chain. Preceding modules have dropped their rates, and succeeding finish tasks in higher rate than they are produced by the encoder. This behavior is reflected as a straight time line but exponential CPU usage.

### 5.4 Multi stream performance

In previous experiments, only one DVB-T stream was used and the parameters were set to maximize its throughput. However, DVB-T process chain is linear and therefore if there
is a slow module it will delay every module in the chain. To make sure that all resources are in use, in this experiment multiple DVB-T streams are used. Even if one stream is blocked, the others can keep the resources busy. Fig. 5.5 shows the average throughput per stream for Calypso(GPU) and GNU Radio(CPU). Parallel streams are 1,2,3,4,5,6,7 respectively.

In Calypso, the maximized throughput setup has already caused GPU congestion. Additional streams have their tasks scheduled. Because all the streams have the same priority, average throughput per stream is dropped by half every time a new stream is added, but the aggregated throughput remain the same. This can also be verified by the CPU usage, which does not change significantly regardless of the number of streams. The reason is that the CPU has finished the orchestration and tasks waste a lot of time in waiting queues within the GPU. This behavior resembles a kernel with high process time. But instead of being processed, the tasks spend long periods waiting.

On the other edge, GNU Radio(CPU) starts with low throughput for one stream, but as the streams are increasing the aggregated throughput tends to exceed that of Calypso(GPU). Contrary to the commodity GPU, the high-end CPU which was used gives to GNU Radio(CPU) the advantage of multiple cores with higher clock. As a result, as new streams are appended, they are still plenty of resources to be used.

What should be noted here, is that Calypso performs only a bit lower than GNU Radio for multiple streams, using resources with a cost almost of one eighth of the resources used by GNU Radio (Advise comparison table on introduction Chapter).

![Figure 5.5: Average throughput per stream drops in Calypso(GPU) and GNU Radio(CPU) as the number of parallel DVB-T streams increasing](image-url)
5.4.1 Multi streams to batch size performance

The previous experiment showed that the chosen batch size value has already congested the GPU from the first DVB-T stream. Intuitively, someone could assume that if smaller batch size was used the GPU would not be congested.

This experiment presents the correlation between batch size and number of DVB-T streams to the output throughput per stream. The selected batch sizes are 525, 1050, 1575, 2100, 3150 and the parallel streams 1, 2, 4 and 8.

Fig. 5.6 shows that the buffer size does not change the behavior dramatically. Starting with smaller batch size (right line), there is lower throughput with higher CPU usage due to higher orchestration. Higher batch sizes, require less orchestration. However in every case the behavior is the same. As more streams are added, the per stream performance drops by half. This is due to a combination of GPU saturation and GPU occupancy.

![Figure 5.6: Average throughput per stream tends to be unaffected by the batch size, as the number of DVB-T streams increasing](image-url)
Chapter 6

Epilogue

6.1 Conclusions

Calypso is a batch streaming engine that combines CPU for task parallelism and GPU for data parallelism. CPU entities called modules are used as containers for the GPU workers. Adjacent modules exchange data by exchanging buffer descriptors over a concurrent queue. The upper (coordinator) part of a module is executed on a dedicated CPU thread while the lower (labor) part is executed on a dedicated CUDA stream in the GPU.

The engine is dedicated to minimize the end-user intervention by automatically handling the asynchronous operation, synchronization and pipelining of individual modules. Further, it takes care of operational duties like data exchange and memory management, so that the module-developer can be focused on the functionality part.

Contrary to other CPU engines like GNU Radio, it is designed for GPU environment and take into account the batch processing, coalesced data transfers and asynchronous streams.

Its versatile design allows it to be used evenly in high performance computing with throughput-demands and real time computing with time constrains.

The comparison of GPU implementation managed by Calypso and CPU implementation managed by GNU Radio showed that Calypso perform twice as better than GNU Radio, with a GPU cost at 1/8 of the CPU. However, when multiple DVB-T are executed simultaneously, the GNU Radio(CPU) implementation performs slightly better than Calypso(GPU). This occurs because the high-end, 24-core CPU is saturated slower than the used GPU.

6.2 Future work

To improve Calypso, more work has to been done in two axis. First, new topologies are to be added. Specifically the conditional branching and recursion, which are not supported in any contemporary framework. Additionally, modules have to be modified to support feedback control. This could open the way to configurable topologies like on-demand branch priority.
Bibliography


Chapter 7

Appendix

7.1 C++11

7.1.1 Variables

An lvalue (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue.]

An xvalue (an “eXpiring” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references. [Example: The result of calling a function whose return type is an rvalue reference is an xvalue.]

A glvalue (“generalized” lvalue) is an lvalue or an xvalue.

An rvalue (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object or subobject thereof, or a value that is not associated with an object.

A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue]

We will focus on the lvalues and rvalues which are used in the constructors and we will reason in detail the implementation decisions.

7.1.2 Type Inference

Enables the programmer to spend less time having to write out things the compiler already knows. On top of that, with the advent of template types and template metaprogramming techniques, the type of something, particularly the well-defined return value of a function, may not be easily expressed. C++11 allows this to be mitigated in two ways. First, the definition of a variable with an explicit initialization can use the auto keyword.
7.1.3 References
A reference is a simple reference datatype that is less powerful but safer than the pointer type inherited from C.
C++11 adds a new nonconst reference type called an rvalue reference, identified by `T&&`. This refers to temporaries that are permitted to be modified after they are initialized, for the purpose of allowing "move semantics".

7.1.4 Lambda functions
C++11 provides the ability to create anonymous functions, called lambda functions. These are defined as follows:

```cpp
1 | [](int x, int y) -> int { return x + y; }
```

7.1.5 Smart Pointers
A smart pointer is an abstract data type that simulates a pointer while providing additional features, such as automatic memory management or bounds checking. These additional features are intended to reduce bugs caused by the misuse of pointers while retaining efficiency. Smart pointers typically keep track of the memory they point to. They may also be used to manage other resources, such as network connections and file handles.

A unique_ptr is a container for a raw pointer, which the unique_ptr is said to own. A unique_ptr explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the std::move function can be used to transfer ownership of the contained pointer to another unique_ptr. A unique_ptr cannot be copied because its copy constructor and assignment operators are explicitly deleted.

A shared_ptr is a container for a raw pointer. It maintains reference-counted ownership of its contained pointer in cooperation with all copies of the shared_ptr. The object referenced by the contained raw pointer will be destroyed when and only when all copies of the shared_ptr have been destroyed.

A weak_ptr is a container for a raw pointer. It is created as a copy of a shared_ptr. The existence or destruction of weak_ptr copies of a shared_ptr have no effect on the shared_ptr or its other copies. After all copies of a shared_ptr have been destroyed, all weak_ptr copies become empty.

7.1.6 Structures
Unordered Maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys. In an unordered map, the key value is generally used to uniquely identify the element, while the mapped value is an object with the content associated to this key. Types of key and mapped value may differ.

Map Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.
In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key
and mapped value may differ, and are grouped together in member type value type, which is a pair type combining both:

```cpp
typedef pair<const Key, T> value_type;
```

Internally, the elements in a map are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

map containers are generally slower than unordered map containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

### 7.1.7 Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

#### Variadic Template

Prior to C++11, templates (classes and functions) could only take a fixed number of arguments, which had to be specified when a template was first declared. C++11 allows template definitions to take an arbitrary number of arguments of any type.

```cpp
template<typename... Values>
class tuple;
```

Variadic templates may also apply to functions, thus not only providing a type-safe add-on to variadic functions (such as printf) - but also allowing a printf-like function to process non-trivial objects.

```cpp
template<typename... Params>
void printf(const std::string &str_format, Params... parameters);
```

#### Variadic Function

Similar to Variadic template, a variadic function can take an arbitrary number of arguments. Prior to C++11 similar functionality could be provided using va structure. An example comparison is depicted in 7.1.7.
// Prior to C++11
int add_nums(int count, ...)
{
    int result = 0;
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        result += va_arg(args, int);
    }
    va_end(args);
    return result;
}

// C++11
template <class ...T>
int add_nums(T &&...args )
{
    std::vector<any> vec = {args...};
    int result = 0;
    for (int i = 0; i<vec.size(); ++i ) {
        result += i;
    }
    return result;
}

Listing 7.1: Variadic function comparison

7.1.8 Tuples

Tuples are objects that pack elements of -possibly- different types together in a single object, just like pair objects do for pairs of elements, but generalized for any number of elements.

Conceptually, they are similar to plain old data structures (C-like structs) but instead of having named data members, its elements are accessed by their order in the tuple.

The selection of particular elements within a tuple is done at the template-instantiation level, and thus, it must be specified at compile-time, with helper functions such as get and tie.

Unpacking

As said before, tuple unpacking is done at compile-time. Knowing at number of arguments and the corresponding types at compile-time is not as straight forward as someone would expect. It is as hard as "writing a program whose parameters will rewrite another program". Using a combination of C++11 features and stackoverflow advices it was possible to implement it using templates as a kind of pre-processor.
Listing 7.2: Tuple unpacking

```cpp
/**
 * Pass the number of elements in tuple to _unpack_tuple
 * along with the tuple itself
 */
template<typename ... QType>
Constructor( std::tuple<QType...>& Queues )
{
    _unpack_tuple( typename gens<sizeof...(QType)>>::type(), Queues );
}

/**
 * The number of elements in tuple (argument 1) results to a sequence
 * of numbers in the range of [0,ELEMENTS_IN_TUPLE]. Using this vector
 * it is possible to access elements at compile time
 */
template<int ...S, typename ... Qin>
void _unpack_tuple( seq< S... >, std::tuple< Qin... >& iQueues)
{
    std::get<S > (iQueues)...;
}

/**
 * These helpers functions take as argument a number N and return a vector
 * of numbers ranging from 0 to N.
 */
template<int ...>
struct seq {}; 

template<int N, int ...S>
struct gens: gens<N - 1, N - 1, S...> 
{
}
template<int ...S>
struct gens<0, S...> 
{
    typedef seq<S...> type;
};
```

7.1.9 Threads

C++11 comes with a core threading library which simplifies a lot the multithreaded programming. Leaving aside details other details, thread instantiates as simple as

```cpp
// Using named functions
void call_from_thread() 
{
    std::cout << "Hello, World" << std::endl;
    std::thread t1(call_from_thread);

    // Using lambda function
    auto t = std::thread( [&] { 
        while(true) func.work();});
```

7.1.10 Thread Locking

Mutual exclusion algorithms prevent multiple threads from simultaneously accessing shared resources.

**Mutex** provides basic mutual exclusion facility.

**Timed Mutex** provides mutual exclusion facility which implements locking with a timeout.

**Unique lock** is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership,
and use with condition variables.

A condition variable is a synchronization primitive that allows multiple threads to communicate with each other. It allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed. A condition variable is always associated with a mutex.

**Conditional Variable** provides a condition variable associated with a std::unique_lock

**Wait** blocks the current thread until the condition variable is woken up

**Notify one** notifies one waiting thread

### 7.1.11 Atomics

The atomic library provides components for fine-grained atomic operations allowing for lockless concurrent programming.

**Atomic_store** atomically replaces the value of the atomic object with a non-atomic argument.

**Atomic_load** atomically obtains the value stored in an atomic object.

### 7.1.12 Constructors

**Default constructor** is a constructor which can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter).

**Copy constructor** is a special constructor for a class/struct that is used to make a copy of an existing instance. If an explicit copy constructor is not declared, the compiler gives you one implicitly. The implicit copy constructor does a member-wise copy of the source object.

**Move constructor** does not allocate new resources. Instead, it pillers other’s resources and then sets other to its default-constructed state. A move constructor enables you to implement move semantics, which can significantly improve the performance of your applications. As a short explanation, rvalues need not to be copied as would have happened with copy constructors and can be moved through the constructor chain. For example in the case of constructor $A(x) \rightarrow constructorB(x) \rightarrow constructorC(x) \rightarrow constructorD(x)$, parameter $x$ will be references (used) along the constructor call path, while with copy constructor it would have been copied 1 time per constructor. It is especially handful for handling rvalues variables.

```cpp
1 C::C();       //C++11 default constructor
2 C::C(C& other);  //C++11 move constructor
```

You can delete special member functions as well as normal member functions (and constructors) and non-member functions to prevent them from being defined or called. Deleting of special member functions provides a cleaner way of preventing the compiler from generating special member functions that you don’t want. The function must be deleted
as it is declared; it cannot be deleted afterwards in the way that a function can be declared
and then later defaulted.
In the case below, by explicitly disabling copy semantics we can use only move semantics
(both constructor and assignment operator) for improving performance and enhancing
security.

```cpp
struct noncopyable
{
    noncopyable() = default;
    noncopyable(const noncopyable&) = delete;
    noncopyable& operator=(const noncopyable&) = delete;
};
```

7.1.13 Inheritance

One of the most important concepts in object-oriented programming is that of inheritance.
Inheritance allows us to define a class in terms of another class, which makes it easier to
create and maintain an application. This also provides an opportunity to reuse the code
functionality and fast implementation time.
When creating a class, instead of writing completely new data members and member
functions, the programmer can designate that the new class should inherit the members of
an existing class. This existing class is called the base class, and the new class is referred
to as the derived class.

Virtual Functions

If a function is advertised as virtual function by the base class, then the derived class
can optionally provide an alternate implementation that will be used instead. Base class
implementation is only used as the default if not derived implementation is provided.
On the other hand, if a function is advertised as pure virtual function by the base class,
then the derived class is obligated to provide an implementation for this function.

7.1.14 Rvalue wrappers

Because Rvalue handling is a rather new feature is not widely supported by and in some
cases quite "hacks" must be used. The most important wrappers being used are listed
bellow.

std::ref are similar to normal references ("&") of the C++ language. Wrapper references
are useful above all for function templates, where references to parameters rather
than copies are needed.

std::move takes an object and allows you to treat it as a temporary (an rvalue). Although
it isn’t a semantic requirement, typically a function accepting a reference to an rvalue
will invalidate it. When you see std::move, it indicates that the value of the object
should not be used afterwards, but you can still assign a new value and continue
using it. In formal, it converts an rvalue to an xvalue.

std::forward has a single use case: to cast a templated function parameter (inside the
function) to the value category (lvalue or rvalue) the caller used to pass it. This
allows rvalue arguments to be passed on as rvalues, and lvalues to be passed on as
lvalues, a scheme called "perfect forwarding."
std::forward_as_tuple Constructs a tuple of references to the arguments in args suitable for forwarding as an argument to a function. The tuple has rvalue reference data members when rvalues are used as arguments, and otherwise has lvalue reference data members.

7.2 CUDA

CUDA (after the Plymouth Barracuda), which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

7.2.1 Kernel

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

In the example below, we present two codes implementing the same functionality one time running in CPU and one in CUDA.

7.2.2 SIMT vs SIMD

SIMT and SIMD both approach parallelism through broadcasting the same instruction to multiple execution units. This way, you replicate the execution units, but they all share the same fetch/decode hardware.

If so, what’s the difference between "single instruction, multiple data", and single instruction, multiple threads"? In NVIDIA’s model, there are 3 key features that SIMD doesn’t have:

- Single instruction, multiple register sets
- Single instruction, multiple addresses
- Single instruction, multiple flow paths

In the examples above, we can see that in the typical CPU case scenario a hello function is being executed sequentially and the contents of b are summed to the contents of a (element-wise).
In the cuda case, the same is happening but in parallel. Instead of executing the hello function `blocksize` times, it is executed once but with `blocksize` instances. Each instance operates on a different memory offset (`threadIdx.x`) thus no locking is needed for this case.

### 7.2.3 Memories

Cuda relies on a hierarchy of memories trading capacity, access time and scope. As depicted below, we can see general characteristics of each memory and how they are connected to thread indexing.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Size</th>
<th>Latency</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>On-Chip</td>
<td>16,384 32-bits registers per SM</td>
<td>0</td>
<td>R/W</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>On-Chip</td>
<td>16 KB per SM</td>
<td>0</td>
<td>R/W</td>
</tr>
<tr>
<td>Constant</td>
<td>On-Chip</td>
<td>64 KB</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>Texture</td>
<td>On-Chip</td>
<td>3 GB</td>
<td>&gt; 100 cycles</td>
<td>R</td>
</tr>
<tr>
<td>Global</td>
<td>Off-Chip</td>
<td>3 GB</td>
<td>&gt; 100 cycles</td>
<td>R/W</td>
</tr>
<tr>
<td>CPU Motherboard</td>
<td>Off-Chip (over northbridge)</td>
<td>4 GB (mobo dependent)</td>
<td>Northbridge dependent</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Figure 7.1: Cuda Memories

![Figure 7.1: Cuda Memories](image1)

![Figure 7.2: Cuda Memory Hierarchy](image2)

### 7.2.4 Streams

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently.

**Synchronization**

Since all operations in non-default streams are non-blocking with respect to the host code, you will run across situations where you need to synchronize the host code with operations in a stream

```c
cudaDeviceSynchronize()  // blocks the host code until all previously issued operations on the device have completed.
```
cudaStreamSynchronize(stream) can be used to block the host thread until all previously issued operations in the specified stream have completed.

cudaEventSynchronize(event) act similar to their stream counterparts, except that its result is based on whether a specified event has been recorded rather than whether a specified stream is idle. You can also synchronize operations within a single stream on a specific event using cudaStreamWaitEvent(event) (even if the event is recorded in a different stream, or on a different device!).

Cases

Synchronous

1. For transfers from pageable host memory to device memory, a stream sync is performed before the copy is initiated. The function will return once the pageable buffer has been copied to the staging memory for DMA transfer to device memory, but the DMA to final destination may not have completed.

2. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.
3. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.

4. For transfers from device memory to device memory, no host-side synchronization is performed.

5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

**Asynchronous**

1. For transfers from pageable host memory to device memory, host memory is copied to a staging buffer immediately (no device synchronization is performed). The function will return once the pageable buffer has been copied to the staging memory. The DMA transfer to final destination may not have completed.

2. For transfers between pinned host memory and device memory, the function is fully asynchronous.

3. For transfers from device memory to pageable host memory, the function will return only once the copy has completed.

4. For all other transfers, the function is fully asynchronous. If pageable memory must first be staged to pinned memory, this will be handled asynchronously with a worker thread.

5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.