Providing Support for the Movidius Myriad1 Platform in the SkePU Skeleton Programming Framework

by

Rosandra Cuello

LIU-IDALITH-EX-A--14/058--SE

2014-11-04
Final Thesis

Providing Support for the Movidius Myriad1 Platform in the SkePU Skeleton Programming Framework

by

Rosandra Cuello

LIU-IDA/LITH-EX-A--14/058--SE

2014-11-04

Supervisor: Lu Li
Examiner: Christoph Kessler
Abstract

The Movidius Myriad1 Platform is a multicore embedded platform primed to offer high performance and power efficiency for computer vision applications in mobile devices. The challenges of programming multicore environments are well known and skeleton programming offers a high-level programming alternative for parallel computing, intended to hide the complexities of the system from the programmer. The SkePU Skeleton Programming Framework includes backend implementations for CPU and GPU systems and it has the capacity to support more platforms by extending its backend implementations.

With this master thesis project we aim to extend the SkePU Skeleton Programming Framework to provide support for execution in the Movidius Myriad1 embedded platform. Our SkePU backend for Myriad1 consists of a set of macros and functions to compose the different elements of a Myriad1 application, data communication structures to exchange data between the host systems and Myriad1, and a helper script and auxiliary files to generate a Myriad1 application.

Evaluation and testing demonstrate that our backend is usable, however further optimizations are needed to obtain good performance that would make it practical to use in real life applications, particularly when it comes to data communication. As part of this project, we have outlined some improvements that could be applied to obtain better performance overall in the future, addressing the issues found with the methods of data communication.
Acknowledgements

I would like to thank my examiner Christoph Kessler and my supervisor Lu Li for their time and advice. I would also like to thank the Movidius support team, particularly Cristian Olar, for answering my questions and giving me their time when I visited their offices. Finally, I am thankful for the financial support I received from the EU FP7 project EXCESS to travel to Timisoara, Romania during the implementation of my project.
## Contents

1 Introduction ........................................ 1  
   1.1 Motivation ...................................... 1  
   1.2 Project Goal .................................... 2  
   1.3 Project Approach ............................... 2  
   1.4 Project Limits .................................. 2  
   1.5 Thesis Outline .................................. 3  

2 Programming Multicore Architectures .............. 4  
   2.1 Parallel Programming Models .................. 5  

3 Skeleton Programming and SkePU .................... 8  
   3.1 Algorithmic Skeletons ......................... 8  
   3.2 SkePU Overview ................................ 9  

4 Overview of the Myriad1 Platform ................. 14  
   4.1 Leon Processor .................................. 14  
   4.2 SHAVE Processors ............................... 15  
   4.3 Memory and Input/Output Subsystems ........... 16  
      4.3.1 CMX Memory .................................. 16  
      4.3.2 The Serial Peripheral Interface .......... 17  
   4.4 The Myriad Build System ....................... 17  
   4.5 MoviDebug and MoviDebugServer ............... 18  

5 Porting SkePU to Myriad1 ............................ 19  
   5.1 Application distribution ....................... 19  
   5.2 Data Size Considerations ...................... 21  
   5.3 Host Application ............................... 22  
      5.3.1 Loading a Myriad Application ............. 22  
   5.4 Myriad Application .............................. 23  
      5.4.1 Leon Code ................................... 23  
      5.4.2 SHAVE Code ................................ 24  
      5.4.3 Shared Memory ............................... 25  
      5.4.4 File Generation Script ..................... 26  
   5.5 Data Communication ............................. 27
CONTENTS

5.6 Execution Flow of a Skeleton Call .................................. 27

6 Evaluation ........................................................................ 29
  6.1 Time Distribution ......................................................... 29
  6.2 Computational Performance .......................................... 32
    6.2.1 Map ................................................................. 33
    6.2.2 Reduce ............................................................ 34
    6.2.3 MapReduce ....................................................... 34
    6.2.4 MapArray ......................................................... 34
    6.2.5 MapOverlap ..................................................... 34
    6.2.6 Scan ............................................................... 35
  6.3 Power and Energy Measurements .................................... 36

6 Limitations and Possible Optimizations ................................. 39
  6.1 Application Loading Overhead ....................................... 39
  6.2 Communication Overhead ............................................. 41
  6.3 Memory Limitations .................................................... 41
  6.4 Smart Containers and Memory Management ..................... 42

8 Related Work .................................................................... 44
  8.1 Algorithmic Skeletons and SkePU ................................... 44
  8.2 Programming Embedded Multicore Systems ..................... 45

9 Conclusions and Future Work ............................................ 46
  9.1 Conclusions ............................................................. 46
  9.2 Future Work ............................................................ 47

Bibliography ....................................................................... 49

Appendices ........................................................................ 53

A Abbreviations .................................................................. 54

B Modified and Added Files ................................................ 56
Chapter 1

Introduction

1.1 Motivation

Parallelism has been used for decades to extract performance from computations. Historically, this exploitation of parallelism had mainly been accomplished through the development of micro-architectural features that take advantage of instruction-level parallelism, such as pipelining and superscalar, or through the use of compiler techniques, such as VLIW architectures. Additionally, the performance of microprocessors was continually improved with growing clock frequencies.

By the early 2000s the increasing clock frequencies were resulting in unsustainable levels of power consumption and heat dissipation and manufacturers began to offer general-purpose computer systems and embedded devices with several computing cores running at lower clock speeds. This development was pressed by the need to maintain performance growth in the face of the limitations of single-core systems.

Today there are a great variety of multicore processor implementations, but unlike single-core systems, there is no unified parallel programming model to program these architectures. One of the most difficult aspects of programming for multicore architectures is exploiting the parallelism available in the system. Writing programs to exploit this parallelism is often more difficult than writing sequential programs for single-core systems. This challenge extends from general desktop programming to embedded programming.

The Movidius Myriad1 Platform is a multicore embedded platform optimized to offer high performance for computer vision applications in mobile devices while maintaining power efficiency. However, the Myriad1 platform brings a lot of the difficulties present in many multicore embedded devices. Although Movidius offers a good set of programming tools for their first generation platform, a great deal of work is required to write a simple application and the learning curve is steep.
Higher-level programming abstractions might help increase the programmability of the system. Skeleton programming offers a high-level programming model for parallel computing, intended to hide the complexities of the system from the programmer. The SkePU Skeleton Programming Framework already includes backend implementations for CPU and GPU systems and it has the capacity to support more platforms by extending its backend implementations.

In this master thesis we explore the possibilities to provide support for the Movidius Myriad1 Platform in the SkePU Framework. Our interest is to show the potential of the SkePU framework by extending it to support a greater quantity and variety of platforms.

1.2 Project Goal

With this master thesis project we aim to extend the SkePU Skeleton Programming Framework to provide support for execution in the Movidius Myriad1 embedded platform. This support consists of compatible implementations for the existing data-parallel skeletons and container types. The primary requirement of this project is to obtain a working implementation of a SkePU backend for the Movidius Myriad1 platform.

1.3 Project Approach

For this project we were working with two main elements, the SkePU Skeleton Programming Framework and the Movidius Myriad1 platform. The first step was to familiarize ourselves with the main conceptual factors behind SkePU and its practical implementation details to identify the elements that would need to be adapted for a different hardware platform. Simultaneously, we needed to understand the relevant details of the Myriad1 platform, mainly through the technical documentation provided by Movidius.

The next step was to analyze the specifics of both these elements to determine what was the best way to make them work together. The Myriad1 system is not designed to work easily with general-purpose computers, which presented some interesting challenges.

The resulting backend is evaluated using some synthetic benchmarks, which mainly rely on applications written for existing backends. The evaluation exposes some limitations in our implementation, mostly caused by hardware and platform restrictions. We consider these limitations and examine some possible approaches towards solving them.

1.4 Project Limits

This project only implements the data-parallel skeletons in SkePU. We did not attempt to implement the Farm task-parallel skeleton, as it requires sup-
port for the StarPU runtime system, which the Movidius Myriad1 Platform does not have.

It is important to mention that some of the platform constraints noted throughout this report are valid only for Movidius’ first generation hardware, Myriad1. While we were working on this project Movidius has been developing the Myriad2 platform, its next generation of hardware. Myriad2 will differ from Myriad1 in several aspects some of which will be highlighted as they come into attention. The characteristics of the Myriad2 should be assessed separately in the future and it is one of our main recommendations that the next step for this project be to extend support to this new platform.

1.5 Thesis Outline

The thesis is outlined as follows:

- Chapter 2: Programming Multicore and Embedded Multicore Architectures. This chapter presents a short look at the variety of multicore architectures and the programming models used for them.

- Chapter 3: Skeleton Programming and SkePU. This chapter outlines the concepts and motivations behind skeleton programming and describes the SkePU Skeleton Programming Framework.

- Chapter 4: Overview of the Movidius Myriad1 Platform. This chapter covers details of the Movidius Myriad1 Platform, with focus on the view of the system from a programmer’s perspective.

- Chapter 5: Porting SkePU to the Myriad1 Platform. This chapter presents the implementation of our SkePU backend for the Myriad1. It represents the body of work that is the result of this project.

- Chapter 6: Evaluation. This chapter covers the evaluation of the SkePU backend for the Myriad1 Platform.

- Chapter 7: Limitations and Possible Optimizations. This chapter covers in more detail some limitations identified during the evaluation process and presents possible optimizations to reduce the impact of these limitations.

- Chapter 8: Related Work. This chapter discusses related work.

- Chapter 9: Conclusions and Future Work. This chapter discusses the overall results of this project. It also presents future work that could be done to improve and expand on what is presented in this thesis work.
Chapter 2

Programming Multicore and Embedded Multicore Architectures

The use of multiple cores operating at lower frequencies has been a popular solution to the limits of single-core processors for a long time. For embedded applications, multicore Systems-on-Chip (MCSoCs) are becoming the preferable solution for several applications and design problems. The reduction of overall power consumption and heat generation that results from the use of lower speeds is particularly compelling for embedded systems. There is presently a vast selection of multicore CPUs and graphics processing units (GPUs) on the desktop computer market, as well as many MCSoC solutions in the embedded computing market.

Multicore systems can be broadly categorized as homogeneous or heterogeneous [20]. A heterogeneous system contains multiple cores with different characteristics, features, and instruction sets. In contrast, homogeneous multicore devices implement multiple identical cores.

The main differences found in current multicore implementations are the setup of the cores, their interaction, their level of coupling, their communication infrastructure and topology, their memory sub-system architecture, and their associated programming model. The cores typically interact via shared memory, message passing, direct-memory access (DMA) transfers, or via SIMD (single-instruction multiple-data) access to a global memory.

Current multicore devices include CPUs (like IA32, x86-64, PowerPC, SPARC, or IA64), massively parallel GPUs, heterogeneous multicore platforms like the Cell Broadband Engine, user-configurable Field Programmable Gate Arrays (FPGAs), coprocessor boards like ClearSpeed’s line of products, and hybrid platforms like the Convey HC1 hybrid core. Current approaches and technological solutions comprise heterogeneous and hierarchical archi-
2.1. PARALLEL PROGRAMMING MODELS

As a result of the great diversity of devices and platforms, each with its own characteristics and features, an equally great diversity of programming approaches and environments can be found. Because devices generally differ in the configuration and arrangement of functional and control units, and in the organization of data flow from main memory to the compute cores, the instruction sets and the generic or vendor-specific programming concepts differ as well. In some programming approaches, only parts of the hardware structure are exposed to the user and are accessible by the programmer. The flexibility of the system and control given to the programmer varies, ranging from full automatization of internal processes to complete control and responsibility.

Programming models present an abstraction of the capabilities of the hardware to the programmer. Because it is an abstraction, a programming model is not tied to a specific platform, but rather, it applies to all platforms that adhere to the particular abstraction. In some cases, more than one programming model can be applied to a single platform.

Algorithms and applications contain varying degrees of parallelism. A good parallel programming model should be able to exploit all available types of parallelism to maximize performance. Four basic forms of parallelism, with varying degrees of granularity can be found – bit-level parallelism, instruction-level parallelism, data-level parallelism and task-level parallelism.

- **Bit-level parallelism.** Bit-level parallelism extends the hardware architecture to operate simultaneously on larger data by increasing the processor word size. This form of parallelism is generally transparent to the programmer.

- **Instruction-level parallelism.** Instruction-level parallelism (ILP) consists on identifying instructions that do not depend on each other and can thus be executed in parallel. ILP can be exploited by processors and compilers with little involvement from the programmer. Hardware techniques such as instruction pipelining, super-scalar, and out-of-order execution, as well as compiler supported optimizations are routinely used to exploit this form of parallelism [11].

- **Data-level parallelism.** Data-level parallelism allows multiple processing units to process data concurrently. Unlike bit-level and instruction-level, this form of parallelism is not generally exploitable without programmer intervention and the programming model employed plays a major role. Iterative structures, such as for-loops, are usually good sources of data-level parallelism.

- **Task-level parallelism.** Task-level parallelism distributes different applications, processes, or threads to different processing units, and it relies on the programmer to identify the parts of the application that may be executed independently.

Out of these four types of parallelism, multicore programming focuses the most on data-level parallelism and task-level parallelism. Programming models vary in their support of exploiting one or the other, some are strongly focused on task-level parallelism and provide abstractions which are flexible enough to also exploit data parallelism, while others are mainly focused on the exploitation of structured or loop-level parallelism and provide only weak support for irregular and dynamic parallelism at the granularity of tasks [11].

Parallel programming models can also be classified by the memory abstraction they present to the programmer. In **shared-memory models** all execution entities, be it processes or threads, have access to a common memory uniformly accessible to all. In turn, **distributed-memory models** assume that no physical shared memory exists in the system and the execution entities must communicate in a different way, mainly by message-passing.

Among the shared-memory programming models we can find POSIX threads (Pthreads) and OpenMP. Pthreads is an extension of the POSIX standard for the Unix family of operating systems [40]. Pthreads is very low level and programs are generally considered hard to develop, debug and maintain. OpenMP is a set of compiler directives and library functions which are used to parallelize sequential C/C++ or Fortran code [37].
Currently OpenMP is considered the *de facto* standard for shared-memory programming. OpenMP has been successfully implemented for a number of embedded platforms [11].

Message-passing is the dominating model for programming distributed-memory systems. The message-passing model allows for the communication between execution entities via the exchange of explicit messages. The Message Passing Interface (MPI) is a specification for message-passing operations and it is implemented as a library which can be used by several programming languages, such as C/C++, Java and Fortran [26]. MPI is widely used on computational clusters, there also exist lightweight implementations specialized for embedded systems, such as LMPI [1].

The increasing adoption of hardware accelerators and general-purpose graphical processing units (GPGPUs) has expanded the landscape of parallel programming. The use of these devices create heterogeneous systems, where a host, usually a CPU, off-loads portions of code for them to execute. To program a system like this, the programmer must provide different programs for each part of the system.

There are two dominant models for programming GPGPUs, the Compute Unified Device Architecture (CUDA) and OpenCL. CUDA [36] is a programming model developed by NVIDIA and it is specifically targeted to its own GPUs. CUDA is implemented as an extension of the C language. OpenCL [18], in contrast to CUDA, is an open standard that is meant to be platform independent. It is maintained by the Khronos Group. OpenCL is designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. As such, it is not intended only for GPU devices, but for a wide array of processors, such as CPUs, FPGAs and digital signal processors (DSPs).

Hybrid programming models can be obtained when combining some of the previously mentioned models in the same program. Combinations such as Pthreads/MPI and OpenMP/MPI are popular when targeting clusters of multicores/multiprocessors [11]. Another possibility is represented by the Partitioned Global Address Space (PGAS) languages. PGAS treats memory as a globally shared entity which is logically partitioned, with each portion being local to a processor [45].

There are other parallel programming models that are not categorized based on memory abstraction. Several language extensions geared towards facilitating parallel programming exist, two examples being Cilk and Sequoia. Cilk is a language extension for C, developed at MIT and licensed to Intel, which adds support for parallel programming based on tasks or Cilk procedures [27]. Sequoia is a language extension to a subset of C. In Sequoia tasks are represented with a special language construct and are isolated from each other in that each task has its own address space and calling a sub-task or returning from a task is the only means of communication among processing elements [13].
Chapter 3

Skeleton Programming and SkePU

3.1 Algorithmic Skeletons

Parallel programing tends to be more complex in structure than sequential programming. The traditional parallel programming models discussed previously, such as shared-memory and message-passing, often interleave all computation, interaction and communication tasks, making it difficult to create portable programs.

Two distinct but complementary parts can be identified within a parallel program: a computation part and a coordination part. The computation part includes all logic and arithmetic calculations, as well as control flow operations, and can be expressed in a procedural manner. The coordination part consists of communication and interaction tasks that relate to concurrency, such as synchronization, inter-process and intra-process data exchange, and process creation [14].

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication and interaction [14]. The concept of algorithmic skeletons was first introduced by Cole in 1989 [6] with the purpose of defining a programming model which guarantees an efficient implementation in parallel, but abstracts all aspects of coordination, thus appearing non-parallel to the programmer. An algorithmic skeleton (or simply skeleton) is a parameterizable high-level construct modeled after a specific computation or coordination pattern. The internal implementation of the skeleton encapsulates all pattern-specific and platform-specific constructs to provide an interface that can be used to create a program in a similar way to writing a sequential program [9].

Skelelons can be categorized into data-parallel skeletons and task-parallel skeletons:
3.2 SKEPU OVERVIEW

- **Data-parallel skeletons**: These skeletons work by applying operations on bulk data structures. In this instance, parallelism is derived from the application of independent operations to individual data elements.

- **Task-parallel skeletons**: These skeletons work by dividing the problem into different separate tasks. In this instance, parallelism is derived from the existing independence or inter-dependence found between the tasks.

To write a program using skeletons, the programmer first needs to identify the patterns within the desired computation and describe the task using one or more appropriate skeletons. The main program logic, or user function, then has to be described and applied to the skeleton(s). The drawback of this model is that the programmer is restricted to the patterns described by the existing skeleton implementations, and all parts of computations that do not match a predefined skeleton have to be written manually [6]. However, when the computation matches existing skeletons, the advantage comes from the complete abstraction of all lower level and platform-specific details which do not affect the structure of the user function.

Skeleton implementations are often grouped into algorithmic skeleton frameworks. Skeleton frameworks can include several different implementations for each skeleton available, which can be tuned for a specific platform or programming interface [14]. This allows for increased portability and focuses on the description of the algorithmic structure rather than on the details of the underlying hardware architecture.

### 3.2 SkePU Overview

SkePU is a skeleton programming framework with support for multicore CPU and multi-GPU systems. SkePU is implemented as a C++ template library and provides a unified interface for specifying data-parallel and task-parallel computations using skeletons on both multicore CPUs and GPUs. SkePU provides multiple backend implementations of each skeleton: sequential C++ and OpenMP implementations for CPUs, and CUDA and OpenCL implementations for performing computations on existing GPUs in the system [9].

The first version of the SkePU library was presented by Enmyren and Kessler [12], with initial support for data-parallel skeletons on one-dimensional data. The work has been substantially expanded by Dastgeer and Kessler [8, 9, 10], adding support for two-dimensional data containers, improved data management, an offline tuning mechanism for automatic implementation selection, support for the StarPU runtime system, and a task-parallel skeleton.

---

1In computing **backend** denotes an interface or service that is not directly accessible by the user, which performs a specialized function on behalf of a main processor or software system.
The library has three main components: generators for user functions, containers and skeletons. What follows is an overview of the main components of SkePU, as well as an outline of its main features and capabilities.

User Functions
SkePU provides a macro language to specify user functions. A user function expressed using these macros is processed into a form that can be used with the skeletons regardless of the target architecture. The preprocessor macros expand, depending on the target architectures supported, to a structure that contains target-specific versions of the user function. A SkePU user function generated from a macro-based specification is basically a struct containing member functions for whichever implementation can be selected.

Smart Containers
SkePU provides Vector and Matrix containers for storing one-dimensional and two-dimensional data respectively. The interface provided by these containers is very similar to the C++ STL vector container. Instances of these containers are used as operands for the skeleton functions.

SkePU containers are referred to as smart containers because internally they track different (and possibly overlapping) copies of data in different memory units and do memory management for those copies. This memory management allows the framework to optimize data transfers for operands passed to skeleton calls across CPU main memory and GPU device memories [9].

The memory management techniques are implemented specifically for GPU computations to avoid unnecessary transfers between devices and host, in cases where the device data is not immediately necessary on the host. If a skeleton modifies elements of a container in GPU memory, the transfer of the result is delayed until elements are accessed on the host side (e.g., via the \[\] operator). This is called lazy memory copying and can significantly reduce the communication cost if future GPU computations operate on that data without intermediate CPU accesses [9].

Skeletons
Data Parallel Skeletons
SkePU provides six data-parallel skeletons: Map, Reduce, MapReduce, MapOverlap, MapArray and Scan. Each skeleton has a sequential CPU, OpenMP, CUDA and OpenCL implementation. The skeleton functions are represented by objects, also called functors, which overload the () operator to be called like a regular function. A program using SkePU needs to include the SkePU header file(s) for the skeleton(s) and container(s) used in the program.
To instantiate a skeleton, the user function needs to be provided both as a template parameter and as a pointer to an instantiated version of the user function. Listing 3.1 shows a simple application declaring and calling a map skeleton.

A skeleton can be called in two ways. One way is to call it using whole vectors or matrices as operands, applying the operation to all elements in the containers. The other possible way is to call the skeleton using iterators. With this method, a start iterator and end iterator are provided, which makes it possible to apply the operation to a specific section of the container data.

When the skeleton is called, the library decides which implementation to use by using the offline tuning system. However, the implementation can also be selected manually by defining the SKEPU_<backend_name> flag for compilation.

A brief description of each available skeleton follows:

• **Map < f > skeleton.** The Map skeleton produces a result vector (or matrix) \( r \), where every element is computed by a function \( f \) applied to the corresponding elements in one or more input vectors (or matrices) \( v_1 ... v_k \) [9]. The maximum number of operands for this skeleton is currently limited to three.

• **Reduce < \oplus > skeleton.** The Reduce skeleton produces a scalar result \( r \) by applying an associative binary operator \( \oplus \) accumulatively across all elements in an input vector [9]. When applied to a \( R \times C \) matrix, an output vector of size \( R \) or \( C \) is produced for row-wise or column-wise reduction respectively. To compute a scalar result for matrix operands, the skeleton needs to be given two \( \oplus \) operations, one to
apply row-wise and another to apply to the vector resulting from the row-wise reduction. The second $\oplus$ operation can be the same as the first or a different operation.

- **MapReduce $< f, \oplus >$ skeleton.** The MapReduce skeleton is a combination of Map and Reduce operations in the same computation kernel [9]. This skeleton is an optimization that aims to speed up the calculation by avoiding synchronization delays.

- **MapArray $< f >$ skeleton.** The MapArray skeleton is a variation of the Map skeleton. For two vector operands, $v_1$ and $v_2$, each element of the result vector, $r[i]$, is a function of the corresponding element $v_1[i]$ and a unspecified number of elements from $v_2$ [9].

  For one input matrix $m$ and one input vector $v$, in the result matrix $r$ each element $r[i, j]$ is computed as a function of the corresponding element $m[i, j]$ and a unspecified number of elements from $v$ [9].

- **MapOverlap $< f >$ skeleton.** The MapOverlap skeleton is another variation of the Map skeleton. In this skeleton, each element $r[i]$ of the result vector $r$ is a function of several adjacent elements of one input vector $v$ that reside within a certain constant maximum distance $d$ from $i$ in the input vector, where $d$ is a skeleton template parameter [9].

  For a matrix operand, row-wise (or column-wise) overlap is supported where each element $r[i, j]$ of the result matrix $r$ is a function of several row adjacent (or column adjacent) elements of one input matrix $m$ that reside within a certain constant maximum distance $d$ from $j$ (or $i$) in the input matrix. 2D MapOverlap for matrix operands is also supported where each element $r[i, j]$ of the result matrix $r$ is a function of several adjacent elements of one input matrix that reside at a certain constant maximum logical distance from $i, j$ in the input matrix. The parameters $d_r$ and $d_c$ control the maximum row-wise and column-wise logical distance of these elements respectively.

- **Scan $< \oplus >$ skeleton.** The Scan skeleton takes a binary associative function $\oplus$ and computes the prefix-$\oplus$ vector of its input vector $v$ [9]. For a matrix operand, the prefix-$\oplus$ operation can be applied row-wise or column-wise. The most common example of this pattern is the computation of prefix-sums.

**Farm: Task Parallel Skeleton**

The Farm task-parallel skeleton allows for concurrent execution of multiple independent tasks, possibly on different workers. It consists of farmer (also called master) and worker threads. The farmer accepts multiple incoming tasks and submits them to different workers available for execution. The
farmer is also responsible for synchronization and for returning the control back to the caller when all tasks finish their execution.

The workers execute the assigned task(s) and notify the farmer when a task finishes the execution. A task is an invocation of a piece of functionality with implementations for different types of workers available in the system. Moreover, a task could itself be internally parallel (e.g., a data parallel skeleton call) or could be another task-parallel skeleton (e.g., another farm), allowing nested parallel executions [9].

Multi-GPU Support

SkePU supports the simultaneous use of multiple GPU devices present in the system to carry out computations, using either CUDA or OpenCL. If no upper limit is set for the number of GPUs to be used (by setting the SKEPU_NUMGPU flag), SkePU will make use of as many GPU devices as it can find in the system. Work will be divided between the GPU devices based on the computational power of each device, which is determined during the initialization phase of the SkePU environment.

Runtime support for hybrid execution

SkePU also supports hybrid execution, that is, using both CPU and GPU devices in a computation by dividing the work of a given skeleton between all devices. To allow this mode of execution, SkePU includes support for the StarPU task-based runtime system [2] as a possible backend.

When this backend is enabled, skeleton calls in a normal SkePU skeleton program are translated to tasks in the StarPU runtime system. Each task is then executed by a StarPU (CPU, CUDA or OpenCL) worker. The decision of when the tasks are executed is made with the help of a scheduling and selection policy inside the runtime system [9].
Chapter 4

Overview of the Movidius Myriad1 Platform

The Movidius Myriad1 platform is a heterogeneous multi-processor embedded platform mainly geared towards media processing applications. The main goal of the platform is to deliver optimized computer vision applications while maintaining high power efficiency. As an embedded platform, its target setting is mainly mobile connected devices, such as smartphones, tablets, cameras, wearable devices and other various consumer electronic devices.

A Myriad1 device is composed of a 32-bit RISC processor, named Leon, and several SHAVE processor cores, along with a robust memory subsystem, which includes 16/64 megabytes of in-package stacked DDR and 1 megabyte of on-chip RAM (CMX), and a wide range of input/output (IO) peripheral interfaces. The Myriad1 device, the first iteration of the platform, has eight SHAVEs, while the upcoming Myriad2 device brings the count up to twelve SHAVE processors.

This chapter presents an overview of the Myriad1 Platform from the programmer’s perspective. We will note specific details of the Myriad2 in cases where they are relevant for potential improvements of the SkePU library once evaluation hardware becomes available.

4.1 Leon Processor

Leon is a 32-bit microprocessor core. It is based on the SPARC-V8 RISC architecture [39]. The Leon processor implemented on the Myriad1 is a LEON3 variant. The LEON3 core has 64 kilobytes of on-chip LROM and 32 kilobytes of on-chip LRAM dedicated to the Leon processor. The purpose of Leon in the system is to control peripherals, control data flow and manage application state, and it is not meant to be used for intensive computational processing [33].
Leon is the only bootable processor on Myriad1 and is the entry point to the entire system. After performing reset, Leon begins executing from its internal ROM memory. The firmware allows the loading of application code and data from an onboard FLASH chip connected to the Serial Peripheral Interface (SPI) bus. Once the loading process is completed, the bootloader passes control to the loaded application’s Leon entry point. Every Myriad application needs to include a Leon main() function as its entry point, even if it is only to launch a SHAVE core.

The Movidius Myriad Development Kit (MDK) features driver functions and low level components that enable the user to use the Leon processor as a control processor for peripherals and SHAVEs. Leon code compilation and linking is facilitated by the gcc and binutils packages, which are provided with the MDK. These tools are integrated in the MDK build flow, alongside the Myriad1 specific tools. A subset of the standard C libraries is implemented for Leon, but C++ support is still in development and therefore not available.

The Leon processor has complete access to the flat Myriad1 32-bit address space [33]. Leon has separate instruction and data caches. The data cache implements a write-through cache coherency protocol, which becomes particularly important when sharing data between Leon and SHAVEs [31]. The swcLeonUtils library, available from the MDK, provides macros to facilitate reads by Leon on data shared and updated by SHAVEs.

Another detail that requires care when dealing with shared data between SHAVEs and Leon is endianness. The Leon3 processor is implemented as big endian, but the SHAVE cores are little endian, hence sharing any sub 32-bit data types between SHAVEs and Leon will result in seemingly incorrect data. The Myriad1 documentation recommends sharing 32-bit data types only. The Leon4 processor, which is included in Myriad2, is implemented as little endian, thus this concern is eliminated from Myriad2 [34].

4.2 SHAVE Processors

SHAVE stands for Streaming Hybrid Architecture Vector Engine. The SHAVE cores are the main computation elements of the Myriad1 system and are good at performing intensive computational tasks. The architecture of the SHAVE cores is a hybrid between RISC, DSP, VLIW and GPU architectural features. The SHAVE instruction set provides extensive support for vector and integer SIMD (single-instruction multiple-data) operations, advanced address generation capabilities and DMA (direct memory access) [33].

There is Assembly, C and C++ support for programming the SHAVE through the use of moviAsm and moviCompile, both of which are Movidius internally developed tools. It is possible to optimize C/C++ code by including handwritten assembly language code, using either inline assembly or calling the SHAVE assembly code using the calling convention.
The Movidius C/C++ compiler also provides support for vector data types that allow the use of optimized vector SIMD operations, using the Vector Arithmetic Unit (VAU).

Each SHAVE has a dedicated single channel DMA controller that supports DMA tasks between CMX memory and DDR memory [33], which are the most important memory components on the device. The DMA engine allows for good methods of buffering data in order to process it more efficiently. Typically, data would be buffered from DDR to CMX memory chunk by chunk and it would be processed there, and then moved back to output DDR if needed.

A mutex ¹ block is provided to facilitate resource sharing between the different SHAVE cores. The mutex block allows one core to gain exclusive access to a particular resource. The mutex functionality is provided through the svuCommonShave.h header in the MDK.

All SHAVE applications need to have an entry point. SHAVE entry point functions are functions that serve as a starting execution point on one SHAVE. Their symbol is also accessible from the Leon side. The symbol can then be used to take out the address of the starting point and pass it to the swcStartShave* functions, which launch the execution of SHAVE applications from Leon.

4.3 Memory and Input/Output Subsystems

This section covers two elements of Memory and IO subsystems that are relevant to our discussion.

4.3.1 CMX Memory

The CMX memory is comprised of several smaller SRAM blocks. Although CMX memory has 1 megabyte as a whole, it can be seen as 8 ‘slices’ of 128 kilobytes, with each SHAVE having optimized access to its ‘local slice’, both for bandwidth and power consumption. However, slice locality is a weak concept, each slave can still access the other portions of CMX memory at the same cost, but the inter-slice routing resources are finite [33].

For Myriad2 there are 16 slices of 128 kilobytes which are usable by both LEONs, all 12 SHAVEs and hardware computer vision accelerators, making for a total of 2 megabytes. Each of all 12 SHAVEs still has a 128 kilobytes ‘local slice’. The remaining slices are not particularly tied to any SHAVE and may be used for other purposes [34].

¹In parallel or multi-threaded programming a mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.
4.3.2 The Serial Peripheral Interface

The Myriad1 chipset contains three instances of the SPI bus [31]. A SPI bus is a 4-wire serial communications interface. The Myriad1’s SPI bus supports bulk data transfers of up to 25Mbps (Megabits per second).

The SPI ports are configurable as either master or slave. The SPI interface can be accessed and configured through Leon using facilities provided with the MDK.

4.4 The Myriad Build System

The Movidius MDK build system offers the means to build an application and configure it. Several Movidius tools are involved in the build process, including moviAsm, moviCompile, moviLink and moviConvert [32]. The MDK build system contains a suite of available targets in order to build the final executable .elf file.

All MDK projects need to include a Makefile. The main functionality is done already in the common Makefiles that are included with the MDK, but some actions are required in the local Makefile for the project. The Makefile needs to conform to some rules, outlined by the MDK Programmers Guide [33], to provide access to the MDK common build flow settings.

The build flow used in MDK applications involves compiling and assembling source files for both SHAVE and Leon into sparc-elf objects. The SHAVE applications have to be built separately for each project and each project can have one or many SHAVE applications [33]. Each SHAVE can have several applications mapped to it.

A SHAVE application is represented by a .mvlib file, that can then be linked onto any SHAVE core. The build rules for the required mvlibs are filled in for each individual project. The application’s Makefile has to include a build rule for each application. Additionally, the created .mvlib files have to be placed onto the cores. To place applications on cores one needs to add the applications for each SHAVE using .shvXlib names on the Makefile, with X being the SHAVE number.

Certain rules to facilitate symbol sharing and symbol unification must also be included in the application’s Makefile. Shared variables have to be named for each SHAVE in a .symuniq file that is linked to each core. All the specifics can be found in the MDK Programmers Guide provided by Movidius along with the MDK.

The build process produces the final .elf file (Executable and Linkable Format, [41]) for the application and a .mvcmd file [32]. The Executable and Linkable Format is a standard for object files that is used throughout the build flow. An ELF file is used as the final executable object. The .mvcmd file is the boot file for the application. They are interpreted by the code which resides in the Leon ROM, and which is used usually to load and start the code present in the .mvcmd file. The Makefile functional target
‘make flash’ executes the flasher application, which programs the SPI flash memory with the current .mvcmd file [33]. Only one .mvcmd file can be loaded at a time.

4.5 MoviDebug and MoviDebugServer

The Movidius Debugger (moviDebug, [29]) is an application that is used to test and debug code on the Myriad platform. There are several modes available to run moviDebug:

- It can be used as a separate application alongside the moviSim tool, which simulates both Myriad architectures (Myriad1 and Myriad2).
- It can be used as a separate application communicating with the Myriad1 hardware, using the JTAG interface. For this configuration, the user must start moviDebugServer.
- It can be used as a static library, which may be used for integrating moviDebug commands in a C/C++ project in order to allow C/C++ code to interact with the platform. This mode can use several interfaces, including the JTAG (Joint Test Action Group) interface.
- It can be used as a dynamic library for communicating with different targets.

The Movidius Debug Server (moviDebugServer, [28]) is an application that is used to provide a link between moviDebug and the hardware target. To facilitate running moviDebugServer the command ‘make start_server’ is provided as a Makefile functional target.
Chapter 5

Porting SkePU to the Myriad1 Platform

This chapter outlines the extent of our work to extend the SkePU library to provide support for the Myriad1 platform. This work mainly consists of a set of macros to compose Leon and SHAVE code, SHAVE kernel implementations for the data parallel skeletons, data communication functions using the SPI interface, and a helper script and auxiliary files to generate a Myriad application.

In terms of hardware, all development was done using the Mv0153 evaluation board and its accessories provided by Movidius [30], the TotalPhase Aardvark Host Adapter [42] was used for data communication and the host system corresponds to a Linux system. All these components are necessary to compile and run an application using this backend.

5.1 Application distribution

SkePU is a C++ template library. In view of this, support for the C++ programming language was essential for the development of a SkePU backend for the Myriad1 platform. At the time of writing, Movidius does not offer C++ support for the Leon processor, although, while limited, there is good C++ support for the SHAVE cores. Due to the absence of C++ support on Leon, it is not possible to develop and build SkePU applications that use Leon as its main host.

Initially, we explored one solution to build a backend that would run on Leon. The general idea was to use a C++ to C compiler to compile the Leon code, and subsequently compile the generated C code using the Myriad tool chain. The means explored for this purpose was the LLVM compiler infrastructure ([22]). However, the generated C code is very low level and difficult to maintain. Additionally, the C backend required for compilation
was removed from LLVM as of version 3.1 (in May, 2012) [23].

Considering this limitation, we developed the SkePU backend for the Myriad1 platform with a distributed organization in mind. SkePU applications for the Myriad1 are developed for a Host system, which transmits the data to be processed to the Myriad1, using the SPI interface. The Leon processor receives the data, divides it across the SHAVEs and launches the SHAVE cores, which execute the main data processing kernels. Figure 5.1 shows the proposed distribution for the applications.

A SkePU application for the Myriad1 has two components: a main application running on the Host and an application running on the Myriad1 device. Two parts, as was mentioned in the previous chapter, form a Myriad application: a Leon application, which is in charge of control and housekeeping tasks, written in the C language; and a SHAVE application executing the kernel code, written in C++. Each skeleton instance found in the Host application, i.e. each skeleton/user function combination, is represented by an individual Myriad application.

Given the build rules outlined by the Movidius MDK documentation, a SkePU application for the Myriad1 platform requires a file structure that matches the Myriad programming model. Hence, the following file structure is proposed, with some files to be discussed later and excluded for simplicity.

![Proposed Application Distribution](image.png)
5.2. DATA SIZE CONSIDERATIONS

Among the SkePU utilities, a standard *makefile* is provided to compile the host application, along with each Myriad application tied to it.

### 5.2 Data Size Considerations

In an embedded environment like the Myriad1 there is very limited memory space available. The available DDR memory space is 64 megabytes. To avoid memory overflows, 50 megabytes are allocated for the application data (i.e. inputs and outputs of the skeleton). This limits the size of the operands that can be used for skeleton calls, and the maximum size of each operand varies depending on the skeleton. For example, a Map skeleton instance with a ternary user function is limited to $50/4$ megabytes for each operand, including the output; while a Reduce skeleton has almost all 50 megabytes available to its one operand, minus the bytes required by the output element. A possible solution to this limitation is discussed in Chapter 7.

Additionally, the present implementation restricts the allowed data types to 32 bit types, that is, only 32 bit integer and floating-numbers can be used in the calculations. This limitation is caused by the difference in endianness between Leon and SHAVE, which restricts the size of the data that can be shared between them to 32 bits. The difference in endianness is eliminated in the next generation of the platform, the Myriad2. Extending support for the Myriad2 is discussed in the future work section of Chapter 9.
5.3 Host Application

A host application is written in the same way we would write a SkePU application for any of the other existing backend implementations. The code should be compiled with the –DSKEPU_MYRIAD compilation argument to indicate that the Myriad1 backend is to be used. Listing 5.1 shows the code necessary for a host application that is using the Map skeleton.

5.3.1 Loading a Myriad Application

Whenever a skeleton function is called in the host application, the Myriad application corresponding to that skeleton is loaded on the Myriad1. Internally, this is done using the moviDebug static library.

The moviDebug interface is initialized when the host application initializes the SkePU environment, and it remains open until the host application finishes execution. The loading of a Myriad application requires two steps: loading the application .elf file, and starting the loaded application.

In order to successfully load the skeleton applications the host system must be physically connected to the Myriad1 board using the JTAG interface [30]. Additionally, the moviDebugServer application must be running prior to the execution of the host application. If either of these conditions is not met, the application will abort execution with an error message.
5.4. MYRIAD APPLICATION

```
// myriad_app/leon/main.c
#define SKEPU_MYRIAD
#define MAP_SKELETON
#include "skepu/myriad Helpers.h"

SHAVE_APP_ENTRY_POINT_DEF(shave_app)

int main(void)
{
    Setup();
    while(1)
    {
        SHAVE_INVOCATION(shave_app, SK_MAP)
    }
    return 0;
}
```

Listing 5.2: LEON code: providing SHAVE entry points and invoking SHAVE applications

The path to the moviDebug API (application programming interface) and static library must be provided to compile the application. These components are given with the Movidius MDK.

5.4 Myriad Application

As was mentioned above, for each skeleton instance in the host application there is a Myriad application. The applications are compiled along with the host application and the corresponding skeleton application is loaded on Myriad1 and executed whenever a skeleton function is invoked.

Each Myriad application directory must include three components: a Leon application directory, a SHAVE application directory, and a configurations directory, which includes a customized linker script for shared memory configuration.

5.4.1 Leon Code

The Leon application handles three basic things: the initialization and management of the SPI interface, the distribution of the input and output data sections and the launch of the SHAVE cores. This functionality is encapsulated in a set of macros and functions in the library and can be used to express the application in a simple way.

Listing 5.2 presents a LEON source file for a Map skeleton application. All the functions and macros are provided by the skepu/myriad_helpers.h
header. The call to the \textit{Setup()} function handles the initialization of the SPI\textunderscore Slave interface, as well as some miscellaneous initialization tasks.

Two macros are used in this file. The first one declares the SHAVE entry point symbols that are necessary to refer to the applications in the SHAVEs. If these entry point symbols are not included, the SHAVE applications cannot be invoked. The second one is the invocation macro for the SHAVE applications. This macro encapsulates the reading of the input data, the launching of the SHAVEs and the writing back of the output data.

The SHAVE invocation is enclosed by an infinite loop. The purpose of this is to keep the application running and waiting for potential new data for cases where the same skeleton is called several times in a row from the host application. This avoids the need to re-load the same application over again and mitigates the time overhead caused by it, as well as the time overhead caused by initializing all board components. The infinite loop is broken when a different SHAVE application is loaded.

The Leon code is largely the same for all skeletons. Only two things change in every case, the \textit{<skeleton type>} SKELETON flag provided near the beginning of the file and the SK\textunderscore <skeleton type> parameter given to the invocation macro. \textit{<skeleton type>} SKELETON indicates the skeleton type to the preprocessor to select only the pertinent functions. This is done to keep the code size as small as possible by avoiding the compilation of code that would go unused. SK\textunderscore <skeleton type> is used as a parameter for the initialization of input and output data buffers. This is necessary because every skeleton requires a specific data distribution pattern.

Each skeleton differs in the way in which the data is placed and distributed among the SHAVE cores based on the number and length of input and output operands. For instance, the Map skeleton has up to three input operands that need to be distributed among all SHAVEs and returns a structure that has the same size as the inputs, while the Reduce skeleton involves only one input operand and generally a single element as the output. This kind of differences affect the way the data has to be placed in the input and output buffers for each SHAVE and have to be dealt with specifically for each skeleton. Additionally, the type of operand, i.e. vector or matrix, also influences this distribution, however this does not need to be indicated explicitly by the program.

\subsection{SHAVE Code}

A SHAVE application runs on every SHAVE core, in most cases, and it executes the computation kernels. Same as Leon, the \textit{skepu/myriad\_helpers.h} header provides macros that encapsulate the declaration of the entry points, as well as the user function macros, which do not differ from the ones present in the host application.

The entry point macros contain the call to the computation kernels. There is a collection of entry point macros where each macro corresponds to
the appropriate user function and skeleton combination that is being used and which calls the appropriate kernel function. For instance, Listing 5.3 shows a SHAVE source file created for a Map skeleton with a unary user function, as such, the MAP_UNARY_SHAVE_ENTRY_POINT macro must be used. The <skeleton_type>_SKELETON flags must be provided.

Regarding data communication within Myriad1, all data exchanged between Leon and SHAVEs is done using DMA transfers. As a consequence of the write-through cache coherency protocol, the cache must be flushed in Leon before signaling the host to read the output data. Our implementation takes care of this to ensure coherency between what is written by the SHAVEs and what is transmitted back to the host.

5.4.3 Shared Memory

Some skeletons require the use of shared memory, that is, having all cores access the same memory location using the same symbol without having to duplicate data. In the Myriad1, DDR memory and CMX memory are accessible for all processor cores in the device, but the symbols declared in one processor are not shared by default and there are no predetermined shared memory sections. A shared memory section must be fixed for the CMX memory in the application’s (.elf files) memory map.

In order to allocate a shared memory section, the application’s linker script (ldscript) has been customized. When the shared section is appended to CMX memory, it is added immediately after all existing memory sections, which, in this case, is after all data and text (code) sections assigned to the SHAVEs.

The CMX memory is 1 megabyte divided in equal slices of 128 kilobytes for each SHAVE. The data and text sections for a SHAVE occupy 36 kilobytes out of the 128 kilobytes available in its corresponding CMX slice. In a configuration where the ldscript allocates all available SHAVEs (eight in this case), this results in a shared memory section of 92 kilobytes accessible

Listing 5.3: SHAVE code: executing Map kernel and user function

```c
// myriad_app/shave/map.cpp
#define SKEPU_MYRIAD
#define MAP_SKELETON
#include "skepu/myriad_helpers.h"

UNARY_FUNC(square_f, u32, a,
    return a*a;
)

MAP_UNARY_SHAVE_ENTRY_POINT(square_f, u32)
```
CHAPTER 5. PORTING SKEPU TO MYRIAD1

...to all SHAVEs. For most skeletons this configuration works, since they do not need much shared memory. However the MapArray skeleton benefits from larger shared memory space, as this allows it to work on larger input sizes.

To allocate a larger shared memory size, the MapArray skeleton was implemented using only four SHAVEs. To achieve this, the ldscript has been modified to allocate four SHAVEs, numbers 0 through 3, instead of the usual eight SHAVEs. This way, the shared memory section is loaded immediately after the data section of the fourth SHAVE (number 3), allowing for a shared memory section of 512 kilobytes, half of the CMX memory space.

We should note that 512 kilobytes is still rather limited. As a result, only relatively small data sets can be computed using the MapArray skeleton on the Myriad1. This can be potentially improved once the Myriad2 platform is available, since the CMX memory size will be larger.

5.4.4 File Generation Script

A python script to generate the Myriad1 files for each skeleton included in an application was incorporated to the library. This script reads the host application source file and produces the Myriad application files for the skeletons found in it. The files produced have the structures that were outlined above for the Leon and SHAVE applications, as well as the ldscript configuration files.

A collection of pre-defined file structures for each skeleton is provided alongside the script, in the utilities/myriad_files/ directory. The script collects the user function and skeleton declarations from the source file. Using these collected elements and the pre-written files, it determines the skeleton file structure that it needs and the corresponding SHAVE_ENTRY_POINT macro.

To produce the files, the host application has to be given to the script as an argument. As an example, to generate the files for a SkePU program called main.cpp, the following should be executed on the command line:

```
$ python mdk_code_generator.py /home/skepu/utilities/myriad_files main.cpp
```

The script creates a myriad_files directory inside the application’s path, which contains all the Myriad1 applications. The naming convention for the applications is of the form myriad_app_<N>., with N being a number starting at 0. The numbering of the applications is based on the textual order in which the skeletons are placed in the source file.

The script generates the Myriad1 applications based on the order in which the skeleton declarations appear in the file, without any contextual information. Because of this, it is not reliable to generate files for programs where the skeletons are not instantiated in the order in which they appear.
For example, if a function is defined before the `main()` function and instantiates a skeleton, but is called after a skeleton instantiation inside the `main()` function, the static numbering of the application will be incorrect. For cases like this, manual correction of the application directory names should be sufficient to make it work.

The script collects the skeletons in a very simple way by using string matching. The use of static analysis and profiling could be good alternatives to consider contextual information and avoid the problem present when generating files for more complex programs. We discuss this further in the future work section in Chapter 9.

### 5.5 Data Communication

All data transfers between the host and the Myriad1 are done via the SPI interface. In the SPI configuration the host acts as the master and the Myriad1 acts as the slave. Physically, the devices are connected using the TotalPhase Aardvark Host Adapter. The Mv0153 evaluation board requires an attachment numbered Mv0137, also provided by Movidius, to interface with the host adapter.

The host device is connected to the adapter using the USB interface. TotalPhase provides the programming interface on Linux through a shared object library and C/C++ language bindings.

A drawback to this implementation is that the communication channel between the devices is limited by the capabilities of the host adapter. Unfortunately, the connection to the SPI bus on the Mv0153 is restricted to the physical port provided by the Mv0137, which is only compatible with the Aardvark host adapter, and is a hardware limitation that, at this date, cannot be circumvented.

Despite having a theoretical bandwidth of 8 Mbps, we have calculated that with this configuration the host adapter can only reach an average effective transfer rate of 0.48 Mbps. The reasons behind this limitation and possible solutions for it are further discussed in Chapter 7.

### 5.6 Execution Flow of a Skeleton Call

The execution of a SkePU application on the Myriad1 platform involves the interaction between three different applications. The sequence diagram in Figure 5.2 illustrates the application execution flow. In the Host application the skeleton has to be initialized with the user function. Upon initialization the skeleton puts together the name of the Myriad application it is meant to execute when it is invoked in the Host program. The initialization of the skeleton also ensures that both the SPI connection and the moviDebug interface are loaded and ready.
When the skeleton is invoked, the host sends a reset signal and the Myriad application is loaded and executed. As the application begins executing on Leon, several board and component initialization tasks take place, the SPI slave interface is configured and initialized and the application gets ready to receive the input data from the host. The host then sends the input data using the SPI interface. Once the input data is received, Leon partitions the data for each SHAVE core, sets it up on buffers in DDR memory and launches the SHAVEs.

The SHAVEs start executing the kernel function on the data that has been assigned to them, working on it using DMA transfers from DDR to CMX memory. When a SHAVE completes execution it signals Leon that it has finished. Leon waits for all SHAVEs to finish executing and marks the output as ready for the host to read back. The host then reads the output data and the invocation of the skeleton is completed.
Chapter 6

Evaluation

The evaluation of the Myriad1 backend for SkePU was done through the use of synthetic benchmarks, using applications that had previously been written to work with other backends. The execution times were measured to determine the proportion of time dedicated to four components: loading time, waiting time, communication time and computation time. Each skeleton was also evaluated to measure the speedup of computational performance when executing with an increasing number of SHAVE cores. Finally, estimates of power consumption serve as a baseline for future work.

6.1 Time Distribution

The execution times of a Map skeleton function are shown in Figure 6.1. The time has been divided into four components. The communication time corresponds to the time spent on SPI transfers of input data, output data, and parameters between the host and the Myriad1 device. The loading time corresponds to the time spent loading the application to the Myriad1 (with moviDebug). The waiting time corresponds to the time the Myriad1 device spends initializing all requisite components, including the SPI interface, as well as the time it takes the host to query for this status. The computation time corresponds to the time spent executing the computation kernel on the Myriad1.

The skeleton function was executed with a variable number of SHAVE cores, from one to eight. All measurements were taken with an input (and output) size of 1,000,000 elements. The skeleton function was called two times consecutively to show the effects of the method used to mitigate the loading and waiting times. In the graphs, the first bar corresponds to the first skeleton call and the second bar corresponds to the second skeleton call.

In the graph we can observe that the variable number of SHAVEs does not result in a visible difference in execution time on a large scale. In fact, the dominant time factor is the communication time. Figure 6.2 shows a time
CHAPTER 6. EVALUATION

Figure 6.1: Execution times with variable number of SHAVEs.

distribution graph, which demonstrates that in all cases communication time comprises between 97 and 99 percent of the overall execution time, while the loading time, waiting time and computation time represent a small fraction.

Figure 6.3 shows the execution times of the Map skeleton with an exponentially increasing number of elements. As the number of elements grows, the communication overhead becomes more dominant, as can be seen with Figure 6.4. The factors behind this overhead are discussed in more detail in Chapter 7.

Furthermore, we can see that there is a considerable overhead stemming from the loading time of the application. This loading time increases with the quantity of SHAVE cores used, due to the fact that SHAVEs do not share code. Each SHAVE core has to allocate its own program, even when all of them are running the same application, which increases the overall size of the Myriad application. In practice, all applications, except the ones for the MapArray skeleton, are compiled for 8 SHAVE cores. However, this information serves to indicate that larger applications, for instance those that could result from larger user functions, will result in a larger loading overhead.

Figure 6.5 illustrates this more clearly. The size of the SHAVE code was increased by using a longer user function. We can see that as the SHAVE code size progressively increases the loading time grows as well.
6.1. TIME DISTRIBUTION

Figure 6.2: Time distribution with variable number of SHAVes.

Figure 6.3: Execution time with increasing element count.
Another overhead that is somewhat tied to the loading overhead is the waiting time. The waiting time is the time that it takes the board to initialize, coupled with the time it takes the host to query for this status. The waiting time is very constant and it takes a comparable amount of time in all cases, regardless of the number of SHAVEs used or the number of elements being processed. The average waiting time is approximately 1.25 seconds.

As was mentioned in the previous chapter, in cases where the same skeleton is called several times in a row the application remains loaded, in order to mitigate the loading and waiting overheads. The comparison between the running times of a first call of a skeleton versus a second call shows that, in the case of the second call, the impact of the loading time is completely eliminated, while the waiting time is so small that it is almost imperceptible.

These overheads overshadow the time spent on actual useful work and must be addressed further. The next chapter will explain the factors behind these overheads and explore possible optimizations that could reduce its effects.

### 6.2 Computational Performance

To evaluate computational performance we focus on the computation times for each skeleton. The purpose was to analyze the speedup of the computations that can be achieved with an increasing number of SHAVE processors,
6.2. COMPUTATIONAL PERFORMANCE

Figure 6.5: Loading time with increasing user function size

Figure 6.6: Speedup for Map skeleton.

Figure 6.7: Speedup for Reduce skeleton.

without it being obscured by the overheads outlined in the previous section. The speed of the computations was measured directly on the Myriad1 device using the timer functions provided by the MDK. All skeleton functions were executed with 1,000,000 elements of input size, except for MapArray, and the results presented are the average of 5 runs.

6.2.1 Map

The Map skeleton was benchmarked with a factorial function. The function was executed with the same value each time to achieve the same number of operations for all executions.

Figure 6.6 shows the speedup of the map skeleton for 32 bit integers and floating-point numbers. The operation scales almost perfectly in both cases.
6.2.2 Reduce

The Reduce skeleton was benchmarked with a summation function. A summation function performs $N - 1$ operations, where $N$ corresponds to the number of elements in the processed data structure.

Figure 6.7 shows the speedup of the Reduce skeleton for 32 bit integers and floating-point numbers. The operation scales fairly well, but not perfectly. The Reduce implementation includes a synchronization phase where intermediate results are joined and these could account for the overhead.

6.2.3 MapReduce

The MapReduce skeleton was benchmarked with a dot product calculation. This function contains $2N - 1$ operations, where $N$ corresponds to the number of elements in the processed data structure.

Figure 6.8 shows the speedup of the MapReduce skeleton for 32 bit integers and floating-point numbers. The MapReduce skeleton is a combination of Map and Reduce operations. The algorithm scales very similarly to those skeletons.

6.2.4 MapArray

The MapArray skeleton was benchmarked with a matrix-vector multiplication with vector size of 1000 and matrix size of $1000 \times 1000$, which results in $1000 \times (2 \times (1000 - 1))$ operations. The vector of size 1000 is shared between all SHAVEs, while a matrix of size $1000 \times 1000$ is not. The MapArray skeleton is implemented for 4 SHAVE cores, hence it was only measured with up to 4 SHAVEs.

Figure 6.9 shows the speedup of the MapArray skeleton for 32 bit integers and floating-point numbers. The scaling of this skeleton slows down after 3 SHAVEs. Recall that shared memory is allocated in a portion of CMX memory that is not part of each SHAVE’s ‘local slice’, since inter-routing resources are finite, SHAVEs have to wait to be able to access the shared memory section.

6.2.5 MapOverlap

The MapOverlap skeleton was benchmarked with a small convolution kernel. The function performs $10 \times N$ operations, where $N$ corresponds to the number of elements in the processed data structure.

Figure 6.10 shows the speedup of the MapOverlap skeleton for 32 bit integers and floating-point numbers. The MapOverlap skeleton scales very well, as it does not require any form of inter-SHAVE synchronization.
6.2. COMPUTATIONAL PERFORMANCE

6.2.6 Scan

The Scan skeleton was benchmarked with a prefix sum function, which performs $N$ operations, where $N$ corresponds to the number of elements in the processed data structure.

Figure 6.11 shows the speedup of the scan skeleton for 32 bit integers and floating-point numbers. This skeleton scales poorly because it requires two phases of inter-SHAVE synchronization. Intermediate results must be gathered and subsequently added to the results of all other cores. To make these phases clear, Figure 6.12 illustrates the workflow of the scan algorithm.

It is noticeable that when increasing from one to two SHAVEs there is a negative speedup. When executing in only one SHAVE, i.e. in a sequential manner, the processor only sweeps over all the data once. However, when executing with more than one processor, two sweeps over the data are needed instead of one.

So we have that for $p > 1$: 
\[ T_p(n) \approx \frac{2T_1(n)}{p} + T_{synch} \]

where \( p \) corresponds to the number of cores used in the computation, \( T_1(n) \) corresponds to the time it takes to execute the sequential form of the algorithm with \( n \) elements, \( T_{synch} \) corresponds to the time it takes to synchronize the SHAVE cores and \( T_p(n) \) is the time it takes to execute the parallel form of the algorithm with \( p \) cores and \( n \) elements.

When executing with two SHAVEs, that is for \( p = 2 \), we have:

\[ T_2(n) \approx \frac{2T_1(n)}{2} + T_{synch} > T_1(n) \]

Therefore for two SHAVEs we have a decrease in speed as opposed to an increase. As the number of SHAVE cores increases pass two a speedup is experienced, however, because of the need to traverse all the data twice, only approximately 50% of absolute speedup is possible.

### 6.3 Power and Energy Measurements

One of the more important aspects of the Myriad1 architecture is that it is designed to have a small power footprint. While optimizing for power consumption was not a part of our project objectives, it is of interest to demonstrate the power and energy consumption characteristics of our implementation.
6.3. POWER AND ENERGY MEASUREMENTS

This portion was benchmarked with a Map skeleton application executing 100,000 floating-point operations (FLOP) per element on an array of 100,000 elements, that is, 10 billion (10,000,000,000) FLOPs in total. The purpose was to measure the power consumption for an extended period of time with the Watts up? .Net Plug Load Meter [44]., which samples periodically the power consumed by the device directly from the power plug.

The processing kernel of the benchmarked application executes 10 billion FLOPs for a period of 201.4 seconds, which is 49.65 Mflops. It is important to note that this is very far from the Myriad1’s theoretical peak performance of 17.28 Gflops\textsuperscript{1}. This is possibly due to the fact that we are not making use of certain Myriad1 features that would potentially allow our applications to achieve greater performance, mainly the Vector Arithmetic Unit, as it requires the use of special instructions from the Vector Utility Library and further optimization of the data organization. Additionally, the use of these features would need to take place in the user provided functions. Facilitating the use of these features in platform-specific user functions is discussed in the future work section in Chapter 9.

Figure 6.13 shows the power consumption of the application during its execution. During the kernel execution, the average power consumption is 2.79 watts. This translates to a total energy consumption of approximately 562 Joules. By these measurements our implementation has a performance per watt of 17.8 Mflops/watt.

The baseline power consumption provided by Movidius is 2 watts. We

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure613.png}
\caption{Power consumption of application using all eight SHAVEs.}
\end{figure}

\textsuperscript{1}Each SHAVE is able to calculate 12 FLOPs per clock cycle at 180Mhz. By using the formula \textit{cores} \times \textit{clock} \times \textit{FLOPs/cycle} we get $8 \times 180 \times 12 = 17,280Mflops$ or $17.28Gflops$ of theroretical performance.
should highlight the fact that this evaluation setup does not measure the power consumed by the Myriad1 only, but by the entire evaluation board, as the equipment used measures all the power drawn directly from the power plug. This could account for the disparity in the measurements, however, an evaluation setup that measures only the power drawn by Myriad1 would be necessary to confirm this.

Movidius has measured a peak performance per watt of 20+ Gflops/watt for a BLAS GEMM benchmark on Myriad1. Based on our results it is noticeable that careful optimization would be required to allow SkePU applications to take full advantage of the Myriad1 low power capabilities.

There are a wide range of GPU devices with different levels of energy efficiency. For comparison, performance per watt numbers for GPU devices found in research range between 958 Mflops/watt and 5 Gflops/watt [4, 17, 24].
Chapter 7

Limitations and Possible Optimizations

The evaluation allowed us to observe at least two important problems in our current backend implementation, the time overhead caused by loading applications to the Myriad1 and by the data transfer through the SPI interface. This chapter aims to explain the cause of these problems and presents a possible approach to mitigate its effects.

7.1 Application Loading Overhead

The loading of the skeleton applications represents a meaningful portion of the execution time of the skeleton function execution. The time it takes to load an application is a function of the size of the application file. On average it takes 1143376.93 microseconds (1.14 seconds) to load and start an application with the average size of 4.5 megabytes.

This overhead is significant but unavoidable with the current Myriad1 configurations if we want to dynamically load multiple skeleton applications within a host application. We currently attempt to mitigate the loading overhead by keeping the application running on an infinite loop after it is executed for the first time. However, this would only bypass the loading time for applications where a skeleton is called several times, before calling another skeleton. For cases where several skeletons run in an interleaved manner, the application would need to be re-loaded every time.

An alternative we explored to avoid this overhead was to have a single Myriad application containing one Leon application and several SHAVE applications that would correspond to the skeletons present in the host application. In this implementation, the whole Myriad application could be loaded at the beginning of the execution with moviDebug and the Leon application would serve to select the appropriate SHAVE application. In
practice, this solution does not work, because every SHAVE application needs to be loaded on the core from the beginning, the CMX portion that is allocated to code quickly runs out of space and the linking process fails. Preliminary tests for this method yielded memory overloads starting with three or more skeletons in an application, which was unacceptable for our implementation.

We have seen in Chapter 6 that the loading overhead of an application is directly linked to the size of the application file, which is affected by the number of SHAVE cores used. Additional methods to reduce the size of the Myriad application could be studied, such as loading the SHAVE code to one core only and subsequently distributing the code to the other SHAVE’s CMX slices. This would reduce the Myriad application files size, thus reducing the loading overhead.

Reducing the application file size might reduce the overhead, but it does not eliminate it. Here we sketch the basic idea for a solution that could remove the loading overhead through the modification of some elements in the Movidius tool chain.

To execute an application on the Myriad1 without the use of moviDebug, the flasher program is used to upload the mvcmd of the desired application to the flash memory address, where it can be located by the bootloader and executed. The flasher program could be extended to allow the upload of multiple applications, i.e. mvcmd files, to different flash memory addresses. In order to execute the applications alternatively, the bootloader would need to be modified as well.

According to Movidius support staff, it would be possible to modify the bootloader in such a way that it allows for the selection of one entry point among the ones present in flash memory, by providing it with the memory address. The SPI interface could be used to send the signals for the selection of the programs. Both the flasher program and the bootloader’s source code are not freely available to modify. These changes would have to be done with the support of Movidius.

The general idea with this approach is that all the skeleton applications could be compiled and subsequently uploaded to the device using the flasher. The process that in our present implementation consists on loading and running the Myriad application through moviDebug would be replaced by a simple SPI signal selecting the application already in flash memory. This would effectively eliminate most of the loading overhead.

It is important to note that these proposed modifications are in no way trivial. They include the familiarization and modification of source code proprietary to Movidius and their cooperation and support would be necessary to implement them.
7.2 Communication Overhead

The communication overhead is considerable and it dominates over computation time. Unlike the loading overhead, this overhead increases with the size of the input set. The main reason for this is that the Aardvark connector supported by Movidius for SPI data transfers to the Mv0157 board is not optimized for throughput.

In fact, although the official documentation for the connector states that the device has a maximum bit rate of 8 Mbps (Megabits per second), it also clarifies that these bitrates are only achievable within each individual byte transfer and do not extend across bytes. There is a setup time of approximately 9 microseconds required in between each byte, which results in a total transmission period of the byte time plus these 9 microseconds.

Additionally, the connector has a maximum transfer width of $2^{16} - 1$ bytes, which, for arrays that are larger than this, is handled by breaking the transfer down into several smaller chunks. All these factors result in an effective transfer rate of approximately $0.48$ Mbps. This is approximately 17.48 microseconds per byte transferred.

An alternative to achieve a higher throughput is to implement a communication backend with a connector optimized for throughput. The constraint is, of course, hardware compatibility. Movidius only offers an attachment for the Aardvark connector. However, TotalPhase, the company that produces the Aardvark connector, also produces another connector, the Cheeta SPI Host Adapter [43], that is optimized for throughput. The Cheeta has a documented maximum data rate of 30 Mbps.

The Cheeta data sheet details a pin out identical to that of the Aardvark. In theory, with this pin out, this connector should be compatible with the Mv0157 attachment. Both devices, the Cheeta and the Myriad1, have configurable frequencies that could be made to work together.

The SPI transfer rate documented for the Myriad1 is of up to 25 Mbps. It is worth noting that the Cheeta connector does not require setup time between bytes, which means that the transfer rate and frequency of the Myriad1 would be the upper limits for the achievable transfer rate between the devices. The performance gain for the communication could be near a factor of 50 with the maximum configurations.

On the software side, the Cheeta also comes with a documented API that could be interfaced with the SkePU Myriad1 backend with relative ease.

7.3 Memory Limitations

Another important problem is the limitation in memory space, both in CMX memory and DDR memory. These limitations greatly restrict the size of the data sets that can be processed on the Myriad1 platform using our backend implementation.
CHAPTER 7. LIMITATIONS AND POSSIBLE OPTIMIZATIONS

The total size of the skeleton input and output operands is limited by the size of DDR memory in Myriad1, which is 64 megabytes. Of the available 64 megabytes, 50 megabytes are allocated for the input and output operands. To sidestep this limitation, the skeleton implementations could be modified to break their data into several parts, which would individually fit the available memory. The skeleton would then process the data in passes, keeping track of the intermediate results.

The nature of SPI data transfers makes it so that to read data from the device, i.e., to obtain the output from the computation, dummy data, usually 0s, has to be transmitted to trigger the transfer from the slave. It would be potentially beneficial to modify the data transfer functions to transfer useful data instead of dummy data when reading the output. For instance, when reading the resulting output of the first pass of the computation, part of the input data of the second pass could be transmitted. This way the total communication time can also be reduced.

The implementation of the MapArray skeleton encounters a special problem related to the size of CMX memory. One of its operands is meant to reside on shared-memory, which is implemented as a portion of CMX memory accessible to all SHAVEs participating in the computation. The resulting size of the shared-memory section is 512 kilobytes, which effectively restricts the size of the shared operand in a MapArray computation.

A potential solution for this problem is similar to the one proposed above to work around limited DDR memory space. The full shared operand could reside on DDR memory and be passed in smaller sections for the SHAVEs to compute. This would require the SHAVEs to operate on the data and return to Leon to be re-launched with the new shared data, because shared data cannot be transferred via DMA, it can only be copied directly.

Additionally, a method to keep track of the data that is actually in shared-memory while executing the kernel would be necessary. The possible method for this requires further investigation as it is not trivial. To consider this inside the computation kernel would require proper action also inside the user function, which would call for user knowledge and intervention. This would make the user function platform specific and eliminate the abstraction of the platform.

7.4 Smart Containers and Memory Management

SkePU implements smart containers for single-GPU and multi-GPU computations. What makes these containers smart is their capability to internally track different copies of data in different memory units and do memory management for those copies [9]. The benefit of this memory management feature is that data transfers across CPU main memory and GPU device memory can be optimized to avoid unnecessary transfers. A feature similar
to this for our Myriad1 implementation is desirable to improve performance.

Certainly the GPU implementation of smart containers is not directly applicable to Myriad1 because of marked differences in the memory organization. In the Myriad1 backend implementation all operand data is stored contiguously in a 50 megabytes block allocated specifically for it. This data structure needs to be allocated statically at the beginning of the application to make sure that all contiguous space is available. The preset order inside this block is Input 1, Input 2 (if any), Input 3 (if any) and Output. Additionally, this block is needed because a static structure is necessary as a target for SPI transfers. As a result of this memory pattern it becomes more complicated to handle one container’s data without potentially overwriting or otherwise corrupting data that belongs to another container.

The challenge here is the need for one container to be “aware” of the other data on the device that is unrelated to it. Without this awareness, useful data could be overwritten while updating another container, leaving the container that owns said data with outdated knowledge about its state on the device. One possible way to solve this challenge would be to maintain a structure with this information on the device that can be queried by all containers and be modified whenever data is moved.

An initial mechanism could shift existing data on the device and keep track of the positions where the data is moved to in the additional structure. The container itself would need to keep record of at least three things: the position where its data was last loaded on the device (whether it was Input 1, Input 2, Input 3 or Output), the offset of the first element it loaded (considering that computations can be made with partial copies of a container) and the number of elements it previously transferred. By querying the device’s information about where the data is against its own records, the container could locate its data and trigger a copy of it to the proper position for the next computation, alongside transferring any data not on the device, if necessary.

Lazy memory copying could be implemented in a similar way, keeping in mind that from the perspective of the device all data copied to the input positions is read only and results are written only to the output position. If the result of a previous computation needs to be used as an input operand it would have to be moved to the appropriate input position.
Chapter 8

Related Work

8.1 Algorithmic Skeletons and SkePU

The basic idea of skeleton programming was first introduced by Cole [6]. Several algorithmic skeleton frameworks following different programming paradigms have been proposed since then. A good survey of algorithmic skeleton frameworks is offered in [14].

It is worth noting that a parallel skeleton library for embedded multi-cores is presented in [21]. This library is specifically designed for MSoCs and approaches some of the design challenges normally associated with embedded platforms, such as complex memory hierarchies, limited runtime support and the overhead costs of modern programming languages. It is similar to SkePU in that it is implemented as a C++ template library and, like SkePU, has the capacity to support different programming models and hardware platforms.

The library implements two data-parallel skeletons, Par and Hier. The proposed skeletons are implemented using template metaprogramming, with the given rationale for this being the reduction of runtime cost. The authors chose to exploit the application-specific nature of SoCs by using the static information usually hardwired into chips and software components at compile time to customize the executed tasks. To reduce the overhead caused by modern programming languages, they avoid the use of object-oriented features such as function objects and virtual functions.

The Par skeleton parallelizes kernel functions in a way that mimics the CUDA and OpenCL programming models and allows parallel computation on arbitrary dimensions. The Par implementation allows the programmer to write code segments that need to be executed in parallel and the compiler attempts to distribute the computations to the maximum capacity of the underlying hardware. Hier is said to be an enhanced version of Divide-and-Conquer designed to parallelize and specialize the resulting subtasks on the different and more appropriate levels of the memory hierarchy.
Finally, a lot of work has been published on SkePU. It was first presented in [12], where its main motivation, structure and usage is described. It has been further extended in, among others, [8, 9, 10, 25].

8.2 Programming Embedded Multicore Systems

Movidius Myriad1 is an embedded multicore platform. Embedded platforms have a particular set of limitations that, combined with the demands of programming multicore systems, give rise to certain specific challenges [3]. Providing adequate support for programming such systems is becoming increasingly important.

Most embedded platforms are still programmed in C, but C++ is gaining in importance because of its use for multicore programming. However, the code size overhead, produced mainly by object-oriented features of C++, represent a problem for embedded systems, which usually have limited storage. Kuan et al. [19] propose a layered design for the C++ library, with the explicit purpose of controlling code size and facilitating embedded multicore programming.

The challenges involved in porting a C++ library that relies heavily on standard template library (STL) features to a heterogeneous platform are studied in [7]. This article explores the issues present when porting OpenCV, an open-source computer vision library, to Texas Instrument’s embedded platforms. The issues presented are not only relevant to Texas Instrument’s platform but to embedded platforms in general. Two main challenges are discussed: coordination of basic communication between heterogeneous processing cores, and passing large data buffers from one memory space to the other.

An important aspect of programming multicore systems is that low-level details significantly influence performance. The idea of separating low-level details from high-level algorithm descriptions present in skeleton programming is also a part of metaprogramming. Metaprogramming can be used to improve performance portability for parallel systems on chip [16].

An opposing view is suggested in [5], which proposes the use of hardware-aware programming techniques for obtaining good performance from heterogeneous multicore systems, as opposed to elevating the level of abstraction.
Chapter 9

Conclusions and Future Work

9.1 Conclusions

The goal of this master thesis was to obtain a working SkePU backend for the Movidius Myriad1 platform. Evaluation and testing demonstrate that our backend is usable, however further optimizations are needed to obtain good performance that would make it practical to use in real life applications, particularly when it comes to data communication. We have outlined some improvements that could be applied to obtain better performance in the future.

An important challenge was to develop the backend in a way that would hide as much as possible about the underlying platform from the user of the library. To compile and run applications on the Myriad1 we needed a very specific file structure. The need to have extra files aside from the main application calls for the user to be aware of the architecture of the system, at the very least to understand the presence of two types of cores as well as some aspects of the memory organization. This problem was addressed by the inclusion of the file generation script and Makefile to build the application. With the file generation in place, the user is no longer required to know the architecture of the Myriad1 system.

However, it is significant that any given user compiling an application for the Myriad1 platform needs to also have access to the Movidius MDK and documentation, and installation and configuration tasks still need to be performed. In the future this could be further streamlined so that a script can take care of the configurations. As it stands in our current state, the user still has to perform these tasks.

The programming model for the Myriad1 exposes a great deal of low level details and requires a lot of user configurations, which translates to the smallest application requiring a great deal of code. Our approach abstracts
some of the low-level details and simplifies the development of data driven applications for the Myriad1 platform. Notwithstanding, the Myriad1 is at heart an application specific platform, with specific optimizations for the field of computer vision; this complicates the ability of a general-purpose library like SkePU to take advantage of some of its most promising features in terms of performance and cost, such as SIMD operations.

In terms of performance the Myriad1 implementation of SkePU does not surpass the other existing backends. The performance is restricted mostly by the time overheads discussed in Chapters 6 and 7, however, most of the computational performance potential of the device itself is not exploited either. The optimal performance of the device relies on the use of specialized instructions and libraries provided by Movidius, which, while not a negative feature in and of itself, does not lend itself easily to abstraction and generalization.

If the goal of an application is to achieve a certain level of performance, the use of the existing GPU or CPU SkePU implementations would be more appropriate for the given application. The result of this project serves more as a case study for the potential of skeleton programming in general and SkePU in particular to provide support, in terms of high-level abstraction and portability, to a great number of backends, including platforms that are not obvious choices for this kind of applications. Additionally, it serves to demonstrate that an embedded platform such as Myriad1 can be adjusted for general purpose computing even when it is optimized for a specific set of applications.

It is worth noting that the advantage point of the Myriad1 is its optimization for low power consumption. If the SkePU implementation for the Myriad1 platform can be optimized using the suggestions given in this report, as well as other future possibilities, the benefit of using this backend would be more readily seen in the area of energy efficiency. Furthermore, additional work on optimization for low power consumption could take the energy efficiency closer to the peak performance-per-watt capabilities of the platform. This would make the Myriad1 backend a clear choice over GPUs or CPUs in cases where the main goal of the application is not high performance but low power footprint.

9.2 Future Work

The upcoming Myriad2 platform introduces several improvements on the hardware, such as additional memory space and more processing cores. On top of this, the software capabilities and development tools are also being expanded. Adapting these new features to the SkePU backend could potentially increase overall performance. As such, our main recommendation for future work is to analyze the characteristics of the Myriad2 when it becomes available and extend SkePU support for it. The main structure of the library implementation outlined in this work should largely apply for the Myriad2,
but the added capabilities should offer vast improvement.

Implementing the optimizations proposed in Chapter 7 would greatly reduce the impact of the highly time consuming operations of loading applications to the Myriad1 and communicating data via the SPI interface. This solutions would be applicable for both Myriad1 and Myriad2 and should definitely be consider for future work.

One important SkePU feature that was not implemented for this project was lazy memory copying for the smart containers. Given the cost of host to device transfers with the SPI connector, avoiding unnecessary transfers would be beneficial to mitigate the high cost of communication.

In addition, it would be greatly advantageous to allow the users to exploit the Myriad1’s high performance capabilities, such as the special vector and matrix operations offered by the Vector Utility Library and the built-in instruction-set intrinsics. One way to do this could be to extend the SkePU user functions to allow specifying platform-specific versions that could use these facilities.

The file generation script for the Myriad application files is implemented in a very rudimentary way and it encounters problems when the textual order of the skeleton declarations does not match the runtime order in which the skeletons are instantiated. This tool could be improved by using compiler tools to generate the files taking into consideration the dynamic order of the skeleton declarations.

Finally, it would be interesting to explore the possibilities to incorporate a generic interpreter for skeleton calls, as opposed to the current method of loading one application per skeleton. Myriad1 does have the ability to dynamically load SHAVE code, however this method strips all symbol information from the applications. The absence of symbol information removes the ability to share data among SHAVEs, and between SHAVEs and Leon. Sharing data is a requirement for most skeletons, therefore this method was not viable. However, for future work, it would be a good starting point to look at how to develop this feature.
Bibliography


International Conference on Parallel Processing Workshops, pages 65–73. Ieee, September 2010.


Appendices
Appendix A

Glossary of Acronyms and Abbreviations

- BLAS: Basic Linear Algebra Subprograms. A standardized interface for Linear algebra math libraries.
- CPU: Central Processing Unit.
- CUDA: Compute Unified Device Architecture. Parallel programming framework by NVIDIA.
- DDR: Double Data Rate.
- DMA: Direct Memory Access. Memory transfers that do not go through the main processor.
- FLOP: FLoating-point Operation.
- FLOPS: FLoating-point Operations Per Second.
- FPGA: Field-Programmable Gate Array. Integrated circuit designed to be configured by a customer or a designer after manufacturing.
- GCC: GNU Compiler Collection. Compiler system produced by the GNU Project supporting various programming languages.
- GPGPU: General Purpose Graphics Processing Unit.
- GPU: Graphics Processing Unit.
• ILP: Instruction Level Parallelism.
• JTAG: Joint Test Action Group. Common name for the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture
• LMPI: Lightweight MPI.
• LRAM: Low-power Random Access Memory.
• LROM: Low-power Read-only Memory.
• MDK: Myriad Development Kit.
• MPI: Message Passing Interface. Parallel programming model for distributed-memory systems.
• MSoC: Multicore System-on-Chip.
• PGAS: Partitioned Global Address Space. Parallel programming model.
• POSIX: Portable Operating System Interface. Family of standards specified by the IEEE for maintaining compatibility between operating systems.
• QPI: Intel QuickPath Interconnect.
• RISC: Reduced Instruction Set Computing. A CPU design strategy based on the insight that simplified instruction set.
• SCC: Single-chip Cloud Computer. 48-core research chip developed by Intel.
• SHAVE: Streaming Hybrid Architecture Vector Engine.
• SIMD: Single Instruction Multiple Data. A class of parallel computers in Flynn’s taxonomy.
• SPARC: Scalable Processor Architecture. A RISC instruction set architecture developed by Sun Microsystems.
• SPI: Serial Peripheral Interface. A synchronous serial data link standard that operates in full duplex mode.
• SRAM: Static Random-Access Memory. A type of semiconductor memory.
• STL: Standard Template Library. A software library for the C++ programming language.
• VAU: Vector Arithmetic Unit.
• VLIW: Very Long Instruction Word. A processor architecture designed to take advantage of instruction level parallelism (ILP).
Appendix B

List of Files Modified and Added to SkePU

include/skepu/generate.h
include/skepu/globals.h
include/skepu/map.h
include/skepu/maparray.h
include/skepu/mapoverlap.h
include/skepu/mapreduce.h
include/skepu/matrix.h
include/skepu/myriad_helpers.h
include/skepu/reduce.h
include/skepu/scan.h
include/skepu/sparse_matrix.h
include/skepu/src/device_myriad.h
include/skepu/src/environment.h
include/skepu/src/environment.inl
include/skepu/src/generate.inl
include/skepu/src/generate_myriad.inl
include/skepu/src/helper_methods.h
include/skepu/src/map.inl
include/skepu/src/map_myriad.inl
include/skepu/src/map_sparse.inl
include/skepu/src/maparray.inl
include/skepu/src/maparray_myriad.inl
include/skepu/src/mapoverlap.inl
include/skepu/src/mapoverlap_myriad.inl
include/skepu/src/mapreduce.inl
include/skepu/src/mapreduce_myriad.inl
include/skepu/src/matrix.inl
include/skepu/src/matrix_myriad.inl
include/skepu/src/matrix_transpose.inl
include/skepu/src/operator_macros.inl
include/skepu/src/operator_macros_myriad.inl
include/skepu/src/reduce.inl
include/skepu/src/reduce_2d.inl
include/skepu/src/reduce_myriad.inl
include/skepu/src/reduce_myriad_2d.inl

56
include/skepu/src/scan.inl
include/skepu/src/scan_myriad.inl
include/skepu/src/skepu_myriad/generate_kernels_myriad.h
include/skepu/src/skepu_myriad/map_kernels_myriad.h
include/skepu/src/skepu_myriad/maparray_kernels_myriad.h
include/skepu/src/skepu_myriad/mapoverlap_kernels_myriad.h
include/skepu/src/skepu_myriad/mapreduce_kernels_myriad.h
include/skepu/src/skepu_myriad/myriad_leon_macros.inl
include/skepu/src/skepu_myriad/reduce_kernels_myriad.h
include/skepu/src/skepu_myriad/map_kernels_myriad.h
include/skepu/src/skepu_myriad/skepu_myriad.h
include/skepu/src/skepu_myriad/globals.h
include/skepu/src/sparse_matrix.inl
include/skepu/src/vector.inl
include/skepu/src/vector_myriad.inl
include/skepu/vector.h
utilities/Makefile
utilities/mdk_code_generator.py
utilities/myriad_files/generate_files/Makefile
utilities/myriad_files/generate_files/config/myriad1Custom.ldscript
utilities/myriad_files/generate_files/config/myriad2Custom.ldscript
utilities/myriad_files/generate_files/leon/app_config.c
utilities/myriad_files/generate_files/leon/app_config.h
utilities/myriad_files/generate_files/leon/main.c
utilities/myriad_files/generate_files/shave/main.cpp
utilities/myriad_files/generate_files/shave/rules.mk
utilities/myriad_files/generate_files/shave/shave_app0.symuniq
utilities/myriad_files/generate_files/shave/shave_app1.symuniq
utilities/myriad_files/generate_files/shave/shave_app2.symuniq
utilities/myriad_files/generate_files/shave/shave_app3.symuniq
utilities/myriad_files/generate_files/shave/shave_app4.symuniq
utilities/myriad_files/generate_files/shave/shave_app5.symuniq
utilities/myriad_files/generate_files/shave/shave_app6.symuniq
utilities/myriad_files/generate_files/shave/shave_app7.symuniq
utilities/myriad_files/generate_files/shave/sources.mk
utilities/myriad_files/map_files/config/myriad1Custom.ldscript
utilities/myriad_files/map_files/config/myriad2Custom.ldscript
utilities/myriad_files/map_files/leon/app_config.c
utilities/myriad_files/map_files/leon/app_config.h
utilities/myriad_files/map_files/leon/main.c
utilities/myriad_files/map_files/shave/main.cpp
utilities/myriad_files/map_files/shave/rules.mk
utilities/myriad_files/map_files/shave/shave_app0.symuniq
utilities/myriad_files/map_files/shave/shave_app1.symuniq
utilities/myriad_files/map_files/shave/shave_app2.symuniq
utilities/myriad_files/map_files/shave/shave_app3.symuniq
utilities/myriad_files/map_files/shave/shave_app4.symuniq
utilities/myriad_files/map_files/shave/shave_app5.symuniq
utilities/myriad_files/map_files/shave/shave_app6.symuniq
utilities/myriad_files/map_files/shave/shave_app7.symuniq
APPENDIX B. MODIFIED AND ADDED FILES

utilities/myriad_files/map_files/shave/sources.mk
utilities/myriad_files/maparray_files/Makefile
utilities/myriad_files/maparray_files/config/myriad1Custom.lds
utilities/myriad_files/maparray_files/config/myriad2Custom.lds
utilities/myriad_files/maparray_files/leon/app_config.c
utilities/myriad_files/maparray_files/leon/app_config.h
utilities/myriad_files/maparray_files/leon/main.c
utilities/myriad_files/maparray_files/shave/main.cpp
utilities/myriad_files/maparray_files/shave/rules.mk
utilities/myriad_files/maparray_files/shave/shave.app0.symuniq
utilities/myriad_files/maparray_files/shave/shave.app1.symuniq
utilities/myriad_files/maparray_files/shave/shave.app2.symuniq
utilities/myriad_files/maparray_files/shave/shave.app3.symuniq
utilities/myriad_files/maparray_files/shave/shave.app4.symuniq
utilities/myriad_files/maparray_files/shave/shave.app5.symuniq
utilities/myriad_files/maparray_files/shave/sources.mk
utilities/myriad_files/mapoverlap_files/Makefile
utilities/myriad_files/mapoverlap_files/config/myriad1Custom.lds
utilities/myriad_files/mapoverlap_files/config/myriad2Custom.lds
utilities/myriad_files/mapoverlap_files/leon/app_config.c
utilities/myriad_files/mapoverlap_files/leon/app_config.h
utilities/myriad_files/mapoverlap_files/leon/main.c
utilities/myriad_files/mapoverlap_files/shave/main.cpp
utilities/myriad_files/mapoverlap_files/shave/rules.mk
utilities/myriad_files/mapoverlap_files/shave/shave.app0.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app1.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app2.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app3.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app4.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app5.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app6.symuniq
utilities/myriad_files/mapoverlap_files/shave/shave.app7.symuniq
utilities/myriad_files/mapoverlap_files/shave/sources.mk
utilities/myriad_files/mapreduce_files/Makefile
utilities/myriad_files/mapreduce_files/config/myriad1Custom.lds
utilities/myriad_files/mapreduce_files/config/myriad2Custom.lds
utilities/myriad_files/mapreduce_files/leon/app_config.c
utilities/myriad_files/mapreduce_files/leon/app_config.h
utilities/myriad_files/mapreduce_files/leon/main.c
utilities/myriad_files/mapreduce_files/shave/main.cpp
utilities/myriad_files/mapreduce_files/shave/rules.mk
utilities/myriad_files/mapreduce_files/shave/shave_app0.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app1.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app2.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app3.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app4.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app5.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app6.symuniq
utilities/myriad_files/mapreduce_files/shave/shave_app7.symuniq
utilities/myriad_files/mapreduce_files/shave/sources.mk
utilities/myriad_files/reduce_files/config/myriad1Custom.ldscript
utilities/myriad_files/reduce_files/config/myriad2Custom.ldscript
utilities/myriad_files/reduce_files/leon/app_config.c
utilities/myriad_files/reduce_files/leon/app_config.h
utilities/myriad_files/reduce_files/shave/main.cpp
utilities/myriad_files/reduce_files/shave/rules.mk
utilities/myriad_files/reduce_files/shave/shave_app0.symuniq
utilities/myriad_files/reduce_files/shave/shave_app1.symuniq
utilities/myriad_files/reduce_files/shave/shave_app2.symuniq
utilities/myriad_files/reduce_files/shave/shave_app3.symuniq
utilities/myriad_files/reduce_files/shave/shave_app4.symuniq
utilities/myriad_files/reduce_files/shave/shave_app5.symuniq
utilities/myriad_files/reduce_files/shave/shave_app6.symuniq
utilities/myriad_files/reduce_files/shave/shave_app7.symuniq
utilities/myriad_files/reduce_files/shave/sources.mk
utilities/myriad_files/scan_files/Makefile
utilities/myriad_files/scan_files/config/myriad1Custom.ldscript
utilities/myriad_files/scan_files/config/myriad2Custom.ldscript
utilities/myriad_files/scan_files/leon/app_config.c
utilities/myriad_files/scan_files/leon/app_config.h
utilities/myriad_files/scan_files/leon/main.c
utilities/myriad_files/scan_files/shave/main.cpp
utilities/myriad_files/scan_files/shave/rules.mk
utilities/myriad_files/scan_files/shave/shave_app0.symuniq
utilities/myriad_files/scan_files/shave/shave_app1.symuniq
utilities/myriad_files/scan_files/shave/shave_app2.symuniq
utilities/myriad_files/scan_files/shave/shave_app3.symuniq
utilities/myriad_files/scan_files/shave/shave_app4.symuniq
utilities/myriad_files/scan_files/shave/shave_app5.symuniq
utilities/myriad_files/scan_files/shave/shave_app6.symuniq
utilities/myriad_files/scan_files/shave/shave_app7.symuniq
utilities/myriad_files/scan_files/shave/sources.mk
The Movidius Myriad1 Platform is a multicore embedded platform primed to offer high performance and power efficiency for computer vision applications in mobile devices. The challenges of programming multicore environments are well known and skeleton programming offers a high-level programming alternative for parallel computing, intended to hide the complexities of the system from the programmer. The SkePU Skeleton Programming Framework includes backend implementations for CPU and GPU systems and it has the capacity to support more platforms by extending its backend implementations.

With this master thesis project we aim to extend the SkePU Skeleton Programming Framework to provide support for execution in the Movidius Myriad1 embedded platform. Our SkePU backend for Myriad1 consists of a set of macros and functions to compose the different elements of a Myriad1 application, data communication structures to exchange data between the host systems and Myriad1, and a helper script and auxiliary files to generate a Myriad1 application.

Evaluation and testing demonstrate that our backend is usable, however further optimizations are needed to obtain good performance that would make it practical to use in real life applications, particularly when it comes to data communication. As part of this project, we have outlined some improvements that could be applied to obtain better performance overall in the future, addressing the issues found with the methods of data communication.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Rosandra Cuello