A Back-End for the SkePU Skeleton Programming Library targeting the Low-Power Multicore Vision Processor Myriad 2

Sebastian Thorarensen

Supervisor: Nicolas Melot
Examiner: Christoph Kessler
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon 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/](http://www.ep.liu.se/).

© Sebastian Thorarensen
Abstract

The SkePU skeleton programming library utilises algorithmic skeletons to offer a high-level approach for creating parallel applications. By using different back-ends, SkePU applications can run on multicore systems, GPGPU systems, and computer clusters.

Myriad 2 is a low-power multicore vision processor for embedded systems, capable of running parallel applications energy-efficiently. Myriad 2 is heterogeneous, containing two different processor architectures and memories with different characteristics.

In this thesis, we implement a back-end for SkePU, that allows SkePU applications to run on Myriad 2. We describe how the back-end is designed and evaluate the performance of SkePU applications running on Myriad 2. By conducting a series of benchmarks, we show that our back-end achieves enough performance to make SkePU a useful tool for creating applications for Myriad 2. We also show that SkePU applications can run more energy-efficiently on Myriad 2, compared to a GPGPU system.
Acknowledgments

I would like to thank my examiner Christoph Kessler and my supervisor Nicolas Melot for their time, advice, and guidance. I would also like to thank Lu Li at Linköping University, and Brendan Barry at Movidius, for their help during this thesis project. I am also grateful for the financial support I received from the EU FP7 project EXCESS to travel to Heraklion, Greece to present this work at the PDP 2016 conference.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Purpose</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Limitations</td>
<td>3</td>
</tr>
<tr>
<td>2 Parallelism</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Task Parallelism</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Instruction-Level Parallelism</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Data Parallelism</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Abstractions</td>
<td>5</td>
</tr>
<tr>
<td>3 Myriad 2</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Processors</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Memories</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Myriad Development Kit</td>
<td>9</td>
</tr>
<tr>
<td>4 SkePU</td>
<td>10</td>
</tr>
<tr>
<td>4.1 A Complete Example</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Available Skeletons</td>
<td>11</td>
</tr>
<tr>
<td>4.3 Smart Containers</td>
<td>13</td>
</tr>
<tr>
<td>4.4 Previous work on Myriad 1</td>
<td>13</td>
</tr>
<tr>
<td>4.5 Related Work</td>
<td>14</td>
</tr>
<tr>
<td>5 Design and Implementation of the Myriad 2 Back-End</td>
<td>15</td>
</tr>
<tr>
<td>5.1 Build Helpers</td>
<td>15</td>
</tr>
<tr>
<td>5.2 Front-End and Back-End Glue</td>
<td>17</td>
</tr>
<tr>
<td>5.3 Hardware Initialization</td>
<td>17</td>
</tr>
<tr>
<td>5.4 Memory Layout</td>
<td>18</td>
</tr>
<tr>
<td>5.5 Skeleton Setup and Execution</td>
<td>18</td>
</tr>
<tr>
<td>5.6 Skeleton Implementation Code</td>
<td>21</td>
</tr>
<tr>
<td>5.7 Smart Containers</td>
<td>24</td>
</tr>
<tr>
<td>5.8 Vector Operations</td>
<td>24</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>25</td>
</tr>
<tr>
<td>6.1 Simple Skeleton Calls</td>
<td>25</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Myriad 2 processor and memory layout .............................................. 7
5.1 Composition of a SkePU application for Myriad 2 .............................. 16
5.2 Synchronous streaming between DDR and CMX during skeleton execution .......................................................... 20
5.3 Asynchronous streaming between DDR and CMX during skeleton execution .......................................................... 21
5.4 Using gather DMA to copy a submatrix from the DDR to CMX ............. 22
6.1 Binary shift (unary Map) on $10^6$ elements ........................................... 26
6.2 Square root (unary Map) on $10^6$ elements ........................................... 26
6.3 Prefix sums (Scan) on $10^6$ elements ................................................... 26
6.4 Prefix sums with pythagorean addition (Scan) on $10^6$ elements .......... 27
6.5 Binary shift (unary Map) on $10^6$ elements running on MA2450 with different clock speeds .......................................................... 28
6.6 Prefix sums (Scan) on $10^6$ elements running on MA2450 with different clock speeds .......................................................... 28
6.7 Different memory techniques for the Map skeleton ......................... 29
6.8 Different memory techniques for the Generate skeleton ..................... 29
6.9 SIMD speedup for binary shift (unary Map) on $10^6$ elements ........... 30
6.10 SIMD speedup for matrix-vector multiplication (MapArray) with a 3072x3072 matrix .......................................................... 31
6.11 Time difference between executing skeleton implementations using asynchronous DMA streaming, compared to executing skeleton implementations using synchronous streaming, for different user functions, skeletons and number of SHAVE processors. User function 1 performs the least amount of calculations; 5 the most 31
6.12 Execution times using different DMA buffer sizes ............................... 32
6.13 LU decomposition on a 256x256 matrix ............................................. 33
6.14 Median filtering on a 512x512 image (3 bytes per pixel). © 2016 IEEE 33
6.15 SPH fluid dynamics shocktube simulation. © 2016 IEEE ...................... 34
List of Tables

6.1 Energy Efficiency Comparison Between PC and Myriad 2. © 2016 IEEE . . . . . . 34
1 Introduction

In order to overcome problems with limited performance in singlecore uniprocessor systems, parallel computer systems with multiple processors (or multiple processor cores) have become popular. Parallel computer systems can achieve greater performance, at a lower power consumption than singlecore uniprocessor system. It is also often harder and more complex to program parallel computer systems. Programming parallel applications is often more challenging than programming sequential applications, and parallel computer systems often require the programmer to take care of low-level details which can differ between different systems.

Many parallel computer systems used today are also heterogeneous, such as general-purpose graphics processing unit (GPGPU) systems, and embedded systems with digital signal processors (DSP). By utilising two or more different processor architectures, specialising on different tasks, heterogeneous computer systems can achieve greater performance than homogeneous computer systems for many applications. However, heterogeneous computer systems are also often harder and more complex to program, than homogeneous computer systems, as applications for such systems consist of multiple and diverse programming models. Furthermore, heterogeneous computer systems often provide a low level of abstraction, requiring more effort from the programmer.

There exist many tools that help programmers to write applications for parallel computer systems. One such is SkePU, a programming framework which provides an abstraction layer for programming parallel applications, for both homogeneous and heterogeneous systems. SkePU provides an application programming interface (API) with several algorithmic skeletons which with a programmer can construct an application in a way that resembles sequential programming, hiding low-level details of the underlying computer system. SkePU contains several back-ends which allows applications using SkePU to run on different systems, such as multicore computers, GPGPUs, and Myriad 1.

Myriad 1 is a parallel (multicore) system-on-a-chip developed by Movidius Ltd. which can achieve relatively high performance at a very small power consumption. The SkePU Myriad 1 back-end [4], implemented by Cuello, showed that it is possible to support Myriad 1 in SkePU. However, due to limitations in the development tools available for Myriad 1 at the time the Myriad 1 back-end was implemented, the back-end had to be designed in a way that showed to be ineffective. The performance was too low for the back-end to be useful in real-world applications.
Since the back-end’s creation, Movidius has released a new and improved version of the chip called Myriad 2, with improved development tools.

1.1 Purpose

The purpose of this thesis is to create and evaluate a SkePU back-end for Myriad 2. The back-end will allow SkePU applications to run on Myriad 2, and its evaluation will show if SkePU is actually useful in real-world applications for Myriad 2. If shown useful, this back-end will enable SkePU to provide a helpful tool to create applications for Myriad 2, and make Myriad 2 available for existing SkePU applications.

1.2 Limitations

SkePU provides several data parallel algorithmic skeletons, and one task parallel skeleton called Farm. Farm requires support from the StarPU [1] runtime system, which is not available for Myriad 2. Therefore, we limit our implementation to only provide support for SkePU’s data parallel skeletons.
In this chapter we describe different kinds of parallelism in a computer system, and how they correspond to different processor architectures. We also describe what algorithmic skeletons are and how they can be used to create a layer of abstraction, hiding the underlying parallelism in a computer system from the programmer.

2.1 Task Parallelism

A task denotes a sequence of instructions for a computer processor. Task parallelism refers to running two or more tasks in parallel. In a computer, task parallelism can be accomplished by having multiple processors, or multiple processor cores; or by connecting many computers to make a computer cluster.

2.2 Instruction-Level Parallelism

When sequential instructions in a task are independent, they can be executed in parallel without affecting the final result. Executing instructions in parallel is called instruction-level parallelism. Instruction-level parallelism can be controlled both by hardware and software. A common way to gain instruction-level parallelism controlled by the hardware is the use of a superscalar processor; a common way to gain instruction-level parallelism controlled by the software is the use of a very long instruction word (VLIW) processor.

Superscalar

A superscalar processor has the ability to automatically issue several instructions in parallel, without help from the programmer or compiler. When a superscalar processor executes a task, it looks for instructions that can be executed independently on different units in the processor. If two or more independent instructions follow, and these instructions can use different units in the processor, they will be executed in parallel. Many superscalar processors can also reorder instructions to group independent instructions together, to increase the probability that multiple instructions in a task can be executed in parallel.
2.3 Data Parallelism

Very Long Instruction Word (VLIW)

A very long instruction word processor does not itself attempt to parallelize instructions in a task but instead relies on the programmer or compiler to specify which instructions should be executed in parallel. Independent operations that can be executed on different units in the processor in parallel, are grouped together to form a single (very long) instruction. The programmer or compiler has to identify which instructions can be run in parallel in order to exploit the instruction-level parallelism the processor is capable of.

Compared to a superscalar processor, a very long instruction word processor does not need complex logic to find and parallelize independent instructions, as this is handled by the software. This allows a very long instruction word processor to be smaller and more power efficient than a superscalar processor, while still allowing instruction-level parallelism[10]. On the other hand, a very long instruction word compiler needs to be more complex in order to exploit instruction-level parallelism.

2.3 Data Parallelism

Data parallelism refers to working on different data in parallel. Computers with multiple processors, are typically capable of both task parallelism and data parallelism.

A computation that is data parallel, but not task parallel, would be executing a single task on multiple data in parallel. General purpose graphics processing units (GPGPUs) are typically very good at data parallel computation, but less so with task parallel computation.

Single Instruction, Multiple Data (SIMD)

Data parallelism in a single processor can be achieved by the use of Single Instruction, Multiple Data (SIMD). A SIMD processor supports instructions that work on multiple data in parallel. A SIMD instruction could for example be a vector addition. In contrast, a scalar addition is a single instruction, single data (SISD) instruction. In a processor that supports both SIMD instructions and SISD instructions, using the SIMD counter-part, if applicable, will typically be faster.

2.4 Abstractions

When programming for a task and data-parallel computer system, the application must be decomposed and distributed in order to be able to exploit the parallelism. Tasks that can be run concurrently must be identified, and be distributed in the system. Data used by the application must either be copied or shared among the tasks. If the tasks are not fully independent, communication is also needed between the tasks.

Different parallel computer systems may have different characteristics that the programmer must take into consideration when writing an application for the system. For instance, some multi-processor computer systems have uniform memory access (UMA), which means that all processors in the system can access all memory with equal performance. Some other multi-processor computer systems may instead have non-uniform memory access (NUMA), which means that different processors access different areas of the memory with different performance. Some parallel computer systems such as computer clusters may not even have a shared memory between all computers in the cluster, and instead relies on passing messages between the different computers in order to share data.

Different parallel computer systems may also have different programming interfaces. For instance, when programming a parallel application for a typical multi-processor system, the POSIX Threads (pthreads) interface may be used. When instead programming a parallel application for a GPGPU, an interface such as OpenCL or CUDA must typically be used instead.
This means that parallel applications must be modified when moved between computer systems with different interfaces.

**Algorithmic Skeletons**

Cole proposed a way of abstracting decomposition, distribution, data sharing and communication, by the use of algorithmic skeletons. An algorithmic skeleton is a higher-order function which inserts a user-defined sequential function into some pattern. The skeleton provides an abstraction of the underlying implementation of the pattern. If the pattern can be parallelized, the implementation of the parallelization is abstracted as well. The user of the algorithmic skeleton does not need to perform decomposition, distribution, data sharing, or communication, of the pattern herself, nor does she need to concern herself with the characteristics of the targeted parallel computer system and its programming interfaces. Instead the programmer uses a selection of available skeletons to construct an application. Such an application can be moved between all different computer systems supported by the algorithmic skeletons without modification. The skeleton implementation is responsible for making efficient use of the underlying computer system.

When writing an application using algorithmic skeletons, the application is limited by the selection of available skeletons; it must be possible to describe the application with one or more patterns matching the available skeletons.
Myriad 2 is a system-on-a-chip containing multiple processors, memories, hardware accelerators, and external interfaces. Its main application is vision processing, such as video filtering and image recognition, in embedded systems.

Myriad 2 is more power efficient than typical general purpose processors such as those found in PCs. It is capable of 2 trillion 16-bit operations per second at a maximum of 0.5 W \[20\]. The power consumption of a typical general purpose processor running an application, can range from around 20 W to around 250 W \[8\], with a performance of around 0.8 trillion 32-bit operations per second. \[1\]

\[1\] The Intel Xeon E5-2660 v3 is capable of 32 number of 32-bit floating point operations per cycle \[9\]. With 10 cores running at a clock rate of 2.60 GHz, this gives \[10 \cdot 2.6 \cdot 10^9 \cdot 32 = 832 \cdot 10^{12}\] 32-bit operations per second.

\[20\] DDR memory (128 MB)
CMX memory (2 MB)
LEON OS processor
LEON RT processor
L2 cache (256 kB)
L1i (32 kB) L1d (32 kB)
L1i (4 kB) L1d (4 kB)
CMX DMA engine
L2 cache (32 kB)
L1i (2 kB) L1d (1 kB)
L2 cache (256 kB)
L1i (2 kB) L1d (1 kB)
SHAVE processor 0
SHAVE processor 11
CMX memory (2 MB)

Figure 3.1: Myriad 2 processor and memory layout
3.1 Processors

Myriad 2 contains several processors of two different architectures, making it a heterogeneous system: two LEON4 processors called LEON OS and LEON RT, and twelve VLIW vector processors called SHAVE 0-11. Myriad 2 also contains a number of hardware filters. Figure 3.1 shows a diagram of how the different processors and memories are connected in Myriad 2.

LEON

The LEON OS processor in Myriad 2 is used as the boot processor. When the Myriad 2 chip is powered on, the LEON OS processor starts executing an application either from an onboard flash memory, or from an external device. The application running on the LEON OS processor controls the execution of the LEON RT, and the SHAVE processors, which are in a halted state when the Myriad 2 chip is powered on. Typically a real-time operating system or a scheduling application runs on LEON OS, off-loading suitable tasks to the LEON RT and the SHAVE processors.

The LEON RT processor is used to control different multi-media components such as peripherals (like cameras), and the hardware filters. Typically a scheduling application runs on the LEON RT, scheduling work for multi-media components. The LEON RT can also, like the LEON OS processor, control the execution of the SHAVE processors.

SHAVE

SHAVE is a proprietary processor, developed by Movidius, implementing a VLIW architecture. A single instruction word can perform several operations in parallel, as long as the operations use different units in the processor, achieving instruction-level parallelism. In addition, SHAVE supports arithmetic operations on vectors up to 128 bits in length using SIMD. This means that, for instance, four 32-bit floating point numbers can be operated on with a single instruction.

Myriad 2 contains twelve SHAVE processors numbered from 0 to 11. Together these twelve processors can achieve 100 GFlop/s using 32-bit floating point numbers.

Hardware Filters

Myriad 2 contains a number of hardware filters, such as sharpen filter and median filter. It is possible to connect the hardware filters together as a directed acyclic graph, to build a more complex filter. The hardware filters allow for filtering operations to have even greater performance than the SHAVE processors.

3.2 Memories

Myriad 2 contains two different memories. A 128 MB DDR DRAM memory (called the DDR memory) and a faster 2 MB SRAM memory called the CMX memory.

The CMX memory is divided in 16 slices of 128 kB each, numbered from 0 to 15. The first 12 slices have affinity to the 12 SHAVE processors: each SHAVE processor has one slice called its local slice. Slice 0 is assigned to SHAVE 0, slice 1 is assigned to SHAVE 1, and so forth. The bandwidth when accessing non-local slice is limited and accessing non-local slices increases the risk of the processor having to stall to wait for a memory access. Slices 12-15 do not have any particular affinity.
3.3. Myriad Development Kit

DMA

Direct Memory Access (DMA) is a technique which allows data transfers between components and memories independently of the processor(s) in a computer system. With DMA, a processor can initiate a data transfer, and then perform other operations during the transfer. DMA can also lead to faster data transfers as DMA data transfers are not limited by the processing speed of the processor.

Myriad 2 has two DMA engines which handles DMA data transfers: AHB DMA and CMX DMA. AHB DMA provides DMA support for peripherals. CMX DMA provides DMA support between the DDR and the CMX memory; and also supports memory transfers from one location to another in the same memory.

Caches

Myriad 2 contains memory caches for the LEON and SHAVE processors. Each LEON processor has its own set of level 1 and level 2 caches. Each SHAVE processor has its own level 1 cache. One level 2 cache exists, which is shared by all SHAVE processors.

As a difference from the common parallel processor architectures used in PCs, care must be taken by the programmer when sharing data between the different processors on the Myriad 2. The LEON level 1 caches, and the SHAVE level 2 cache, are not automatically kept coherent by the hardware. In addition, the SHAVE level 2 cache implements a write-back policy, which means that the cache must be flushed by the programmer in order for the LEON processors to be able to access data modified by the SHAVE processors.

3.3 Myriad Development Kit

Myriad 2 is programmed with the Myriad Development Kit (MDK). The MDK consists of a compiler for the LEON processors, a compiler for the SHAVE processors, drivers, a build system, a debugger, and several other tools.
SkePU is a skeleton programming library for C++. It contains a number of algorithmic skeletons and supports multicore systems, GPGPU systems, and computer clusters, by the use of multiple back-ends. SkePU can generate sequential code and parallel code using OpenMP, OpenCL, CUDA, and MPI. An explanation of these four interfaces are included in appendix A.

Algorithmic skeletons are implemented as templates, where the templates represent higher-order functions. User-defined sequential functions (simply called "user functions") are passed as template parameters to the skeletons. Depending on the selected back-end, different skeleton implementations will be used.

User functions are defined with preprocessor macros. These macros expands differently depending on the selected back-end. For instance, the macro "BINARY_FUNC" defines a binary function taking two operands, returning a single result.

In order to use a skeleton, the programmer defines a user function and passes it to the skeleton, as shown in Listing 4.2. This creates a new function which is the result of the user function inserted into the algorithm which the skeleton represents. The programmer can then call the newly created function with appropriate operands. The function will be executed on the computer system targeted by the back-end, for instance a GPGPU using OpenCL. The calculations of the function will be divided and executed in parallel, opaquely from the programmers perspective.

SkePU supports three different container types as operands for skeletons: vectors, matrices and sparse matrices. These containers can hold elements of primitive data types, such as integers and floating point numbers; or user-defined data types and structures. These container types are used as operands for skeleton functions, and as results unless the function returns a scalar.

The back-end is selected at compile time by defining an appropriate macro. For instance, defining SKEPU_OPENCL will select the OpenCL back-end when compiling the application. See Listing 4.1 for an example command-line compiling a SkePU application with the OpenCL back-end selected.

**Listing 4.1: SkePU Compilation Example**

```cpp
C++ -DSKEPU_OPENCL -Ipath/to/skepu/include/folder \
application_source.cpp
```
4.1 A Complete Example

Listing 4.2: SkePU Dot Product Example

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus_f, int, a, b,
    return a + b;
)

BINARY_FUNC(mult_f, int, a, b,
    return a * b;
)

int main()
{
    skepu::MapReduce<mult_f, plus_f>
        dotProduct(new mult_f, new plus_f);

    skepu::Vector<int> v1(500, 4);
    skepu::Vector<int> v2(500, 2);

    int r = dotProduct(v1, v2);

    std::cout << "Result: " << r << std::endl;

    return 0;
}
```

Listing 4.2 shows an example application using SkePU. The application calculates the dot product between two vectors, \( v_1 \) containing 500 instances of value 4, and \( v_2 \) containing 500 instances of value 2. When run, the output from the application will be "Result: 4000".

In this example, we first define two binary user functions using the macro `BINARY_FUNC`: `plus_f` which corresponds to addition, and `mult_f` which corresponds to multiplication. These two functions are then passed to the `MapReduce` skeleton, creating a new function called `dotProduct`. We create our two vectors \( v_1 \) and \( v_2 \) and pass them to `dotProduct`, which calculates the dot product between the two vectors, using the selected back-end, and returns the result in \( r \).

If this example were to be compiled with the OpenCL or CUDA back-end selected, on a computer with a GPGPU, the call to `dotProduct` would launch parallel calculations on the GPGPU.

4.2 Available Skeletons

SkePU contains six data parallel skeletons and one task parallel skeleton. The data parallel skeletons are named as follows: Map, Reduce, MapReduce, MapArray, MapOverlap, Scan, and Generate. These skeletons implement algorithms suitable for data parallelization; their operands are vectors or matrices, where the calculations of the elements in the vectors or matrices can be spread among multiple processors and processor cores.

The task parallel skeleton is called Farm, and utilises the StarPU [1] runtime system.
SkePU’s skeletons are explained below.

**Map**

Map applies a user function $f$ of arity $1 \leq k \leq 3$ to each element in $k$ number of vectors $v_k$ each of length $n$, and stores the result in a vector $r$. The mathematical expression of Map on a vector is shown below.

$$r[i] = f(v_1[i], v_2[i], ..., v_k[i]) \forall i \in 0, 1, ..., n - 1$$

Map works in a similar way on matrices.

**Reduce**

Reduce cumulatively applies a binary user function as an operator $\triangle$ to each element in a vector of length $n$, and stores the result in a scalar $r$. The mathematical expression of Reduce on a vector is shown below.

$$r = v[0] \triangle v[1] \triangle ... \triangle v[n - 1]$$

Reduce works in a similar way on matrices. It also supports row-wise and column-wise reductions on matrices, where different user functions can be specified for rows and columns.

**MapReduce**

MapReduce is a combination of the Map skeleton and the Reduce skeleton. It first performs a Map operation with a user function $f$ of arity $1 \leq k \leq 3$, and then performs a Reduce operation with an user operator $\triangle$ on the resulting vector. This results in a scalar $r$. The mathematical expression of MapReduce on a vector is shown below.

$$r = f(v_1[0], ..., v_k[0]) \triangle ... \triangle f(v_1[N - 1], ..., v_k[n - 1])$$

**MapArray**

MapArray is similar to Map, but as an addition, a vector $v_1$ is passed to the user function. The mathematical expression of MapArray on a vector is shown below.

$$r[i] = f(v_1, v_2[i]) \forall i \in 0, 1, ..., n - 1$$

**MapOverlap**

MapOverlap is also similar to Map, but as an addition, it allows the user function to access $l$ number of neighbouring elements in each direction. The mathematical expression of MapOverlap on a vector is shown below. Here, we use “||” as symbol for list concatenation.

$$r[i] = f(v[i - l] || v[i - (l - 1)] || ... || v[i] || ... || v[i + (l - 1)] || v[i + l]) \forall i \in 0, 1, ..., n - 1$$

MapOverlap also supports two-dimensional overlap on matrices, which makes it suitable for image filtering.
4.3 Smart Containers

Scan
Scan performs a prefix sum with a binary user function, used as an operator $\triangle$, on a vector $v$ with length $n$. The result is stored in a vector $r$. The mathematical expression of Scan on a vector is shown below.

$$r[i] = v[0] \triangle v[1] \triangle ... \triangle v[i] \forall i \in 0, ..., n-1$$

Scan works in a similar way on matrices.

Generate
Generate populates a vector $r$ of length $n$, or a matrix $R$ of dimensions $n \times m$, with the results of a user function $f$. The mathematical expressions of Generate on a vector and on a matrix are shown below.

$$r[i] = f(i) \forall i \in 0, ..., n-1$$

$$R[i][j] = f(i, j) \forall i \in 0, ..., m-1, j \in 0, ..., n-1$$

Farm
Farm utilizes the StarPU runtime system to run multiple skeletons in parallel, providing task-parallelism. Any skeleton, both the data-parallel skeletons, and Farm itself, can be used as tasks in the Farm skeleton. It employs dynamic (run-time) scheduling and load-balancing of passed skeletons. Furthermore, Farm allows hybrid execution between different back-ends, so that skeletons for instance can be run in parallel on both the GPGPUs and the CPUs in a GPGPU system.

4.3 Smart Containers

Smart containers [5] is a feature of SkePU which provides automated memory handling in vector and matrix containers when used as operands and results in systems where data must be transferred between different memories or devices, such as systems with one or more GPGPUs. Smart containers ensures that data is synchronized between different memories, and devices, when needed, in a manner that is opaque to the programmer – the programmer of a SkePU application does not need to make any explicit calls to allocate memory or transfer data between devices.

Smart containers uses an optimization called lazy memory copying. Lazy memory copying can reduce the number of redundant memory transfers between subsequent skeleton calls in SkePU applications. For instance on a GPGPU-based system, the result of a skeleton call is not copied back from the GPGPU memory to the computer’s main memory until it is actually accessed by the main application. This avoids unnecessary transfers between the GPGPU memory and the main memory when using subsequent skeleton calls in a pipeline.

4.4 Previous work on Myriad 1

Cuello presented a SkePU back-end [4] for the precursor of Myriad 2: Myriad 1. Myriad 1 differs from Myriad 2 in a number of ways, the most notable being:

- The number of SHAVE processors is 8 instead of 12.
- The number of LEON processors is 1 instead of 2.
• The LEON processor is big-endian instead of little-endian (SHAVE processors are little-endian in both versions).

The Myriad 1 back-end suffers from a number of problems. When the back-end was implemented, the MDK did not have a C++ compiler for the LEON processor. This prevents a SkePU application using the Myriad 1 back-end from running directly on the Myriad 1. Instead a PC host running the SkePU application is connected to the Myriad 1. Skeleton execution is offloaded from the PC host to the Myriad 1’s SHAVE processors during run-time and the LEON processor runs a simple C application which handles communication between the PC host and the SHAVE processors.

The interface between the PC host and the Myriad 1 has a low throughput which causes the communication time to dominate over actual computation time. Furthermore, the Myriad 1 back-end does not make use of the smart containers capability of SkePU, potentially leading to unnecessary communication between the PC host and the Myriad 1 over the slow interface.

Additionally the LEON processor of the Myriad 1 differs in endianness from the SHAVE processors, complicating data transfers between the processors. As a consequence, the Myriad 1 back-end is limited to 32-bit integers and floating point numbers.

4.5 Related Work

There exist multiple other skeleton libraries. Two examples are Marrow [16] and Muesli [2, 7]. Both are C++ template libraries (like SkePU).

Marrow targets GPGPU systems, using OpenCL as a back-end. It provides one data-parallel skeleton, MapReduce. It also provides a number of task-parallel skeletons which, compared to SkePU’s assortment of high-level data-parallel skeletons, provides more flexibility, but at a lower level of abstraction.

Muesli targets multicore and multi-processor systems, computer clusters, and GPGPU-systems, by using OpenMP, MPI, and CUDA as back-ends. It supports a number of task-parallel skeletons, and a number of data-parallel skeletons: Map, Reduce, Scan, and Zip\(^1\).

Neither of Marrow and Muesli contain data-parallel skeletons which allow for element overlap between different user function calls, such as MapOverlap.

There also exist other skeleton libraries specifically targeting multi-core embedded systems. One of them is a C++ template library [13], implementing two skeletons called Par and Hier. Par is a data-parallel skeleton which parallelizes kernel functions, similar to CUDA and OpenCL; and Hier is a task-parallel skeleton which divides tasks into parallelized subtasks, much like divide-and-conquer. Comparing Par to the data-parallel skeletons of SkePU, Par is a more generalized skeleton which leaves more work to be done by user of the skeleton, whereas SkePU provides more specialized and higher level data-parallel skeletons.

Compared to the above mentioned skeleton libraries, SkePU with our Myriad 2 back-end is the only skeleton library that can target a wide range of systems: multi-core systems, GPGPU systems, computer clusters, and embedded systems.

\(^1\)Zip is equivalent to Map with arity 2 in SkePU
5 Design and Implementation of the Myriad 2 Back-End

The Myriad 2 back-end runs entirely on the Myriad 2, without the need for a PC host. The main application runs on the LEON OS processor, and skeletons are loaded to and executed on the SHAVE processors dynamically during run-time. This approach was chosen to avoid some of the problems experienced in the Myriad 1 back-end, stemming from the slow interface between the Myriad board and the PC: slow skeleton loading and data transfers during skeleton execution.

In order to load SkePU applications to the Myriad 2, a PC is connected. However, no communication between the Myriad 2 and the PC is needed during the execution of the application. It is possible for the SkePU application running on the Myriad 2 to communicate with the PC by printing standard output (stdout). Standard output is connected to a terminal on the PC while the SkePU application is running on the Myriad 2.

To better get an understanding of how the back-end is composed, we have divided it into several parts. One of these parts is the build helpers, which helps building SkePU applications for Myriad 2. The rest of these parts consist of source code which makes up the back-end, namely:

- Front-end and back-end glue.
- Hardware initialization.
- Skeleton setup (LEON side).
- Skeleton setup (SHAVE side).
- Skeleton implementations.

Figure 5.1 shows how SkePU, together with the different parts of the Myriad 2 back-end, and the programmer’s code, form an application that can run on Myriad 2.

5.1 Build Helpers

In order to simplify building and running Myriad 2 applications with SkePU, the back-end contains a makefile which performs preparatory actions and configures the MDK build system. The MDK build system is automatically invoked by this makefile; the programmer only needs to either use this makefile directly, or import it in a separate makefile.
5.1. Build Helpers

The Myriad 2 back-end also contains a linker script which describes to the MDK build system where in the memory different parts of the application should be placed.

Example

A SkePU application can be built for Myriad 2 with the command shown in Listing 5.1 and run on Myriad 2 with the command shown in Listing 5.2.

Listing 5.1: Compiling a SkePU application for Myriad 2

```bash
env MAKEFILES=path/to/skepu/include/folder/myriad.mk \
make SKEPU_MYRIAD_SOURCES=application_source.cpp all
```

Listing 5.2: Running a SkePU Application on Myriad 2

```bash
env MAKEFILES=path/to/skepu/include/folder/myriad.mk \
make SKEPU_MYRIAD_SOURCES=application_source.cpp debug
```

The makefile rules “all” and “debug” are rules in the MDK build system which invoke compilation and execution respectively.

Extraction of User Functions and Data Types

The source code of a SkePU application consists of one or more C++ source code files and headers, where one or more files may contain definitions of user functions. This conflicts with a requirement of the MDK: the code for the LEON OS processor, and the code for the SHAVE processors, must be in separate files, as different compilers are used for LEON code and SHAVE code. The user functions must be extracted from the SkePU application source.
5.2 Front-End and Back-End Glue

User functions are defined with the help of preprocessor macros as described in Chapter 4. When compiling a SkePU application for Myriad 2, these macros expand differently depending on the stage in the building process. Before compilation, the user function extraction rule is run, which in turn invokes the C preprocessor on the user’s source code files. When the preprocessor expands a user function definition macro, the line is tagged with a special symbol: ”__skepu_myriad_shave__”. All lines starting with the special symbol are extracted from the SkePU application’s source code and inserted into a separate file, which is compiled together with the skeleton code by the SHAVE compiler in a later stage. When the LEON compiler expands user function macros, they expand normally without being tagged with the special symbol.

SkePU allows the programmer to use her own data structures (defined with the “struct” keyword) as data types for skeleton function operands and results. The definition of these data types must also be extracted and inserted into the same file where user functions are inserted for the SHAVE compiler. To simplify extraction, the SkePU API is extended with a special macro to define data structures. This macro, like the user function definition macros, tags lines with the special ”__skepu_myriad_shave__” symbol when expanded with the C preprocessor during the user function extraction stage, so that the data structure definitions are extracted together with the user function definitions. Listing 5.3 shows an example on how a programmer can define a structure with the macro.

```
// Structure describing a point in two dimensions.
STRUCT(Point,
  float x;
  float y;
)
```

Listing 5.3: Defining a structure with the special macro

5.2 Front-End and Back-End Glue

The front-end and back-end glue consist of code that integrates the Myriad 2 back-end into SkePU. It connects SkePU’s front-end to the part of the back-end which invokes skeletons on the SHAVE processors. When a skeleton function is called, the program control will first reach the SkePU front-end. The SkePU front-end will in turn call the selected back-end, and the back-end will run the skeleton’s calculations on the back-end’s targeted computer system. We refer to the code which connects the Myriad 2 back-end to the front-end as glue.

5.3 Hardware Initialization

Myriad 2 requires the application to initialize the hardware, in order to use many of its features, such as using the SHAVE processors or starting DMA transfers. The Myriad 2 back-end contains the necessary code for hardware initialization. When building a SkePU application for Myriad 2, this code will automatically be linked together with the resulting application. As the function performing the hardware initialization is not called by the programmer’s code, it is added as an “constructor” by annotating the function with the keyword __attribute__((constructor)). The constructor keyword is a non-standard keyword supported by some C and C++ compilers, including the compiler for LEON. Having the hardware initialization function as a constructor will cause it to be called before the programmer’s main function is called.

The hardware initialization function performs a number of tasks, such as setting the clock rate and enabling interrupts. The hardware initialization function for the Myriad 2 back-end
also disables clocks and power supply to components on the Myriad 2 chip which are unused by the back-end, in order to decrease the power usage.

### 5.4 Memory Layout

The main application, running on the LEON OS processor, is loaded into, and run from, the DDR memory. The skeleton code and data that make up the SHAVE applications, is stored in DDR memory. When a skeleton is invoked, its code is copied from its storing area in the DDR memory, to a new, 1 kB aligned area also in the DDR memory. SHAVE applications use virtual addresses for instructions, and the mapping between virtual and physical addresses requires the code to start on a 1 kB boundary. After the code has been copied, the SHAVE application data (such as constants and predefined variables) is copied to each SHAVEs local CMX slice.

### 5.5 Skeleton Setup and Execution

When a skeleton is called, its SHAVE application is first loaded as described in Section 5.4. After loading, the program counter of each SHAVE processor is programmed to point to the loaded code, and one of the integer registers in each SHAVE processor is programmed with a pointer to a structure of arguments called `shaveArgs`. `shaveArgs` contains pointers to the operands, information about operand data types, and information about what user function to insert into the skeleton.

After the SHAVEs have been loaded and configured according to the skeleton currently called, they are started, and start parsing the argument structure `shaveArgs`. When a SHAVE processor has parsed `shaveArgs`, it continues to perform skeleton calculations, each SHAVE working in parallel on a subset of the input operands. After a SHAVE completes its work, it halts. During a skeleton call, the LEON OS processor waits for all SHAVEs to halt, before continuing execution of the main application.

### Communication of Types and User Functions

In order to communicate data types and user functions between the LEON OS processor and the SHAVE processors, each data type and user function is assigned a unique integer id. When a skeleton is invoked, these ids are put in the argument structure `shaveArgs` passed to the SHAVE processors.

On the SHAVE processors, templates are used to generate skeleton code for different data types and user functions. As the ids for data types and user functions are received as variables, they cannot directly be used as template parameters; template parameters must be constants, decidable at compile time. As a solution, template metaprogramming is used to generate conditional calls to the function templates, with different parameters, for all possible values and combination of ids.

### Example

In Listing 5.4 we see an example of how a single operand `input` on the LEON OS processor is stored in the argument structure `shaveArgs`, along with its type id. The pointer of this argument structure is passed to the SHAVE processors as they are started.

```
shaveArgs.input = input;
// getMyriadType maps the type of 'input' to its integer id.
shaveArgs.inputType = getMyriadType<input>::value;
```
In Listing 5.5 we see an example of how a single operand input, along with its type id, is received on the SHAVE processors. The type id is converted to a template parameter with the use of a templated function `callWithTypes`.

Listing 5.5: SHAVE side of receiving a single operand

```cpp
template <InputType>
void calculate(const InputType* input)
{
    // Perform calculations...
}

struct Thunk
{
    template <typename InputType>
    static void call(ShaveArgs* shaveArgs)
    {
        // Cast input to its correct type and pass it to 'calculate'.
        calculate(static_cast<const InputType*>(shaveArgs->input));
    }

    // Convert 'shaveArgs->inputType' to a type and call 'Thunk::call' with
    // this type as an template parameter.
    callWithTypes<Thunk, 1>({ shaveArgs->inputType }, shaveArgs);
};
```

Listing 5.6 shows the expansion of the call to `callWithTypes`. A switch statement is generated which enumerates all valid types, generating a new call for each type.

Listing 5.6: Expansion of `callWithTypes`

```cpp
switch (shaveArgs->inputType)
{
    // Signed and unsigned types are merged into the same call.
    case S8:
    case U8:
        Thunk::call<uint8_t>(shaveArgs);
        break;
    case S16:
    case U16:
        Thunk::call<uint16_t>(shaveArgs);
        break;
    case S32:
    case U32:
        Thunk::call<uint32_t>(shaveArgs);
        break;
    case S64:
    case U64:
        Thunk::call<uint64_t>(shaveArgs);
        break;
    case FP32:
        Thunk::call<float>(shaveArgs);
        break;
    default:
        // Invalid type.
        assert(false);
}
```

As some skeletons use multiple operands, `callWithTypes` supports multiple types in one call. For instance, the call `callWithTypes<Thunk, 2>({ inputType1, inputType2 }, input1, input2)` would recursively generate nested switch statements (as `callWithTypes` is a recursively templated function) to cover all combinations of types.
5.5. Skeleton Setup and Execution

Streaming

Most skeletons stream operand data and results between the slower DDR memory and the faster CMX memory. When streaming is used, operands and results are split into chunks, such that a single chunk can fit into a buffer in the CMX memory. Calculations are performed iteratively on each chunk.

Asynchronous Streaming

Most skeletons use synchronous streaming, but some use asynchronous streaming to improve performance by overlapping data transfers with computations. In some skeletons where streaming is not feasible, the DDR memory is instead accessed through the SHAVE L1 and L2 caches. The skeletons which are using asynchronous streaming also have a corresponding implementation using synchronous streaming, which can be enabled by defining a preprocessor macro. This allows us to evaluate the difference in performance between synchronous streaming, and asynchronous streaming.

Figure 5.2 shows a sequence diagram of (2 iterations of) streaming during skeleton execution for a skeleton using synchronous streaming. We see that the execution of the skeleton must wait for DMA transfers to complete before being able to perform calculations.

Figure 5.3 shows a sequence diagram of (4 iterations of) streaming during skeleton execution for a skeleton using asynchronous streaming. To allow for transfers to overlap computations, two buffers (A and B) must be used. Note that it is possible to queue multiple transfers in order. For instance: one transfer from a buffer in the CMX memory to the DDR memory can be followed by a transfer from the DDR memory to the same buffer in the CMX memory.

The speedup gained from using asynchronous streaming during a skeleton execution, compared to using synchronous streaming, is equal to the amount of time that computations and DMA transfers can overlap in the particular skeleton execution. Figure 5.3 shows a favorable scenario where the computation time is close to the DMA transfer time. Depending
Figure 5.3: Asynchronous streaming between DDR and CMX during skeleton execution

on the user function that is used in a skeleton, it is possible for the computation time to be smaller than the DMA transfer time, which will cause the skeleton to still wait for DMA transfers to complete, like in the synchronous case, reducing the overlap. It is also possible for the computation time to be greater than the DMA transfer time, also reducing the overlap between computations and DMA transfers. In both cases, asynchronous DMA transfers still provide a speedup, as there is still overlap, but the speedup is smaller than the case where the computation time is close to the DMA transfer time.

2D DMA

The CMX DMA engine support gather and scatter operations. The SkePU Myriad 2 back-end utilises this feature when using two-dimensional overlap in the MapOverlap skeleton. Matrices are stored row-wise in the DDR memory. When using two-dimensional overlap in the MapOverlap, submatrices (called “blocks”) of the operand matrix must be transferred to and from the CMX memory. In Figure 5.4, we see an example of how a submatrix is transferred from the DDR memory to the CMX memory using a gather operation. When a submatrix is transferred back from the CMX memory to the DDR memory, a scatter operation is used.

5.6 Skeleton Implementation Code

Listing 5.7 shows how unary Map skeleton is implemented in the SkePU Myriad 2 back-end as a templated C++ function. Synchronous DMA streaming is used in this implementation.
The same skeleton implementation function is run on all 12 SHAVE processors, with the same arguments.
The skeleton implementation function takes a user function, a vector operand and its size; and a result vector where the result is stored, as arguments. In Listing 5.7 we can see how the user function is inserted into the algorithm. We can also see how the vector operand is decomposed into subvectors and distributed among the SHAVE processors. In order for each
SHAVE processor to work on different subvectors of the operand vector, the skeleton implementation code uses a variable containing the number of the SHAVE processor executing the code to differentiate between the SHAVE processors.

5.7 Smart Containers

This section (5.7) uses text from [21], © 2016 IEEE, with permission.

In order to keep the caches on the Myriad 2 coherent, the smart containers feature of SkePU is used. The SHAVE level 2 cache is flushed and invalidated after each skeleton call, but the LEON OS level 1 cache is refreshed on-demand. If a vector that has been modified by the SHAVE processors is accessed by a LEON processor, the vector is refreshed by iterating over the vector and refreshing each element with a special instruction that bypasses the cache.

Some skeletons, for instance MapArray, uses a portion in the CMX memory, which is shared by all SHAVE processors, to store either a vector or a matrix to be accessible by the skeleton’s user function during the entire execution of the skeleton. At the beginning of skeleton call, the vector or matrix to be shared is copied from the DDR memory to the shared portion of the CMX memory. To avoid redundant transfers to the shared memory during subsequent skeleton calls, smart containers are used to keep track of the state of the vector or matrix currently having a copy in the shared memory.

5.8 Vector Operations

This section (5.8) uses text from [21], © 2016 IEEE, with permission.

Movidius’ compiler [17] is capable of automatically generating SIMD instructions for code that is operating on vectors, without the need for any special C extensions or inline assembler. This is called automatic vectorization.

Some skeletons are applicable for automatic vectorization, if they are combined with a user function which contains operations corresponding to available SIMD instructions. To allow the compiler to perform automatic vectorization and generate SIMD instructions for such combinations, user functions are inlined into skeletons, instead of being called. Even if a skeleton is not vectorizable, inlining user functions provides a performance increase as the overhead of function calls is avoided.
To evaluate the performance and functionality of the SkePU Myriad 2 back-end, a number of benchmarks were performed. The benchmarks range from applications designed to test different aspects of the back-end, to already existing applications solving real-world problems. We evaluate specific attributes of the back-end, such as performance scalability when increasing the number of SHAVE processors, and the speedup gained from asynchronous DMA streaming. We also evaluate if the back-end is able to run already existing SkePU applications solving real-world problems and how well these applications scale when increasing the number of SHAVE processors. The power consumption of the Myriad 2 system-on-a-chip was measured for some of the applications, and compared to a PC equipped with a GPGPU running the same applications with the SkePU’s CUDA back-end.

Where not otherwise stated, benchmarks were performed on an MV0182-R4 evaluation board, with a Myriad 2 system-on-a-chip version MA2150, clocked at 600 MHz. Results were received from the chip using a JTAG Debug Connector between a PC and the board.

### 6.1 Simple Skeleton Calls

Figure 6.1 shows the execution times for a unary Map on a vector of $10^6$ elements, performing one binary shift per element, for different number of SHAVE processors. We can note that the execution time for this benchmark scales poorly when using 4 or more SHAVE processors. The reason for this is that the DMA data streaming between the DDR memory and the CMX memory becomes a bottleneck [21]. When this bottleneck is hit, the performance no longer changes in a predictable way when increasing the number of SHAVE processors.

Figure 6.2 shows the execution times for a unary Map on a vector of $10^6$ elements, taking the square root for each element. This benchmark shows noticeably better scaling than the previous example. When performing a more heavy computation per vector element, the DMA data streaming does not become a bottleneck.

Figure 6.3 shows the execution times for a Scan on a vector of $10^6$ elements, calculating the prefix sums of the vector. We can note that using 2 SHAVE processors is slower than using 1 SHAVE processor. The reason for this is that the algorithm behaviour changes when switching from one to multiple processors. The sequential version of the algorithm only needs to access each vector element once; the parallel version needs to access each element twice. After using more than 9 SHAVE processors, the DMA data streaming becomes a bottleneck.
6.1. Simple Skeleton Calls

Figure 6.1: Binary shift (unary Map) on $10^6$ elements

Figure 6.2: Square root (unary Map) on $10^6$ elements

Figure 6.3: Prefix sums (Scan) on $10^6$ elements
6.1. Simple Skeleton Calls

Figure 6.4: Prefix sums with pythagorean addition (Scan) on $10^6$ elements

Similar to the Map benchmarks (figures 6.1 and 6.2), the DMA data streaming bottleneck does not occur when performing a more heavy computation per element. We can see this in Figure 6.4 which shows the execution times of a prefix sum on $10^6$ elements, where addition has been replaced by pythagorean addition.

Why the Bottleneck Occurs

We have two theories on why the DMA data streaming becomes a bottleneck when the computational intensity of each data element is low.

DDR Memory Bandwidth Saturation

One possibility is that the DDR memory bandwidth becomes saturated. The DDR memory on the Myriad 2 chip runs at 533 MHz and is connected to a 32-bit bus. We can calculate the theoretical bandwidth expressed in gigabytes per second with the following expression:

$$\frac{2 \cdot 533 \cdot 10^6 \cdot \frac{32}{8}}{1024^2} \approx 3.97$$

The Map and Scan benchmarks whose execution times are shown in figures 6.1 and 6.3 both run into the DMA data streaming bottleneck. The two benchmarks achieve data transfer speeds (between the DDR and the CMX memory) of at least 2.25 GB/s and 2.37 GB/s, respectively. We calculate this by dividing the amount of data transferred by the total execution time, which gives a lower bound of the data transfer speeds. We can see that the Map benchmark achieves at least 57%, and the Scan benchmark at least 60%, of the theoretical bandwidth. The Programmer’s Guide [19] in the MDK documentation states that a speed of 60%-70% of the theoretical bandwidth is expected. This leads us to believe that the DDR memory bandwidth can be responsible for the bottleneck.

CMX DMA Engine

One other possibility is that too many DMA transfers are issued to the CMX DMA engine. When DMA data streaming is performed, all SHAVEs participating in a skeleton calculation are issuing transfers from and to their local CMX slices. To issue a DMA transfer, a lock must be held while communicating with the CMX DMA engine. When all 12 SHAVEs are issuing transfers, lock contention could possibly decrease performance.

---

1Pythagorean addition is defined as $a \oplus b = \sqrt{a^2 + b^2}$
6.1. Simple Skeleton Calls

The responsibility for issuing DMA transfers could be moved to the LEON OS processor, to reduce lock contention. In addition, gather and scatter operations could be used to send and receive operands and results from all CMX slices in a single transfer, reducing the number of DMA transfers issued. Improving the performance of DMA can however only increase the performance towards the theoretical bandwidth of the DDR memory, not beyond it.

Testing the DDR Memory Bandwidth Saturation Theory

In order to test if DDR memory bandwidth saturation is the underlying reason for DMA data streaming becoming a bottleneck, we have run the Map and Scan benchmarks on a newer version of the Myriad 2 system-on-a-chip: MA2450.

MA2450, in contrast to MA2150 used in the other benchmarks, contains a faster DDR memory. MA2450’s DDR memory runs at 933 MHz, which gives a theoretical memory bandwidth of 6.95 GB/s.

We have tested the Map and Scan applications running into the streaming bottleneck, on three different processor clock speeds: 600, 300, and 100 MHz. 600 MHz is the default processor clock speed. Lowering the processor clock speed increases the relative DDR memory speed when compared to the processors’, as the DDR memory clock speed remains constant.
Figures 6.5 and 6.6 show the Map and Scan benchmarks running on MA2450 at different processor clock speeds. We can see that the DMA data streaming still becomes a bottleneck, even when lowering the processor clock speed.

On MA2450, the two benchmarks achieve data transfer speeds (between the DDR and the CMX memory) of at least 2.80 GB/s and 2.85 GB/s, respectively. We calculate this by dividing the amount of data transferred by the fastest total execution time in figures 6.5 and 6.6 respectively, which gives a lower bound of the data transfer speeds. This is only around 40% of the theoretical DDR memory bandwidth.

This indicates that DDR memory bandwidth saturation is not the reason for DMA data streaming becoming a bottleneck. Our other theory, that of issuing too many transfers to the CMX DMA engine, remains unproven.
6.2 Different Memory Techniques

As explained in Section 5.5, the SkePU Myriad 2 back-end streams data between the DDR memory and the CMX memory, using the CMX memory as a working memory, where applicable. Other methods include transferring data between the DDR memory and CMX memory using a normal memory copy (for instance with the \texttt{memcpy} C standard function); and not using the CMX memory at all, working directly on the DDR memory (through the SHAVEs L1 and L2 caches).

Figures 6.7 and 6.8 show the execution times of two different applications: one using the Map skeleton to perform a binary shift on $10^6$ elements, and one using the Generate skeleton to calculate the Chebychev (chessboard) distances in a $4096 \times 4096$ matrix. Execution times for different memory techniques and different number of SHAVE processors are shown. In the figure, “DMA” denotes using DMA to transfer data between the DDR and the CMX memory, “memcpy” denotes using a normal memory copy to transfer data between the DDR and the CMX memory; and “direct” denotes performing calculations directly on the DDR memory, without transferring data to the CMX memory.

Using DMA transfers between the DDR memory and the CMX memory gives the best performance except in the case where the Generate application is run on only one SHAVE processor, where direct access of the DDR memory gives the best performance. It is likely that the overhead of copying data between the DDR memory and the CMX memory makes direct access of the DDR memory more suitable in cases where memory utilization is low. Running the Generate application on only 1 SHAVE processor causes the time spent on calculations to be high, and the memory utilization to be low.

6.3 Speedup of SIMD

Figures 6.9 and 6.10 show the reduction in execution time from the use of SIMD vectorization, for two different applications: a unary Map performing a binary shift on a vector of $10^6$ elements, and a matrix-vector multiplication on a $3072 \times 3072$ matrix using the MapArray skeleton. In both applications, 32-bit elements are used, which allows each SHAVE processor to work on 4 elements at a time.

When using only one SHAVE processor, we see a 4.4 times speedup in Figure 6.9 (18 to 4.1 ms) [21] and a 3.4 times speedup in Figure 6.10 (241 to 70 ms). When increasing the number of SHAVE processors, the speedup decreases as the DMA data streaming becomes a bottleneck.
Figure 6.10: SIMD speedup for matrix-vector multiplication (MapArray) with a 3072x3072 matrix.

Figure 6.11: Time difference between executing skeleton implementations using asynchronous DMA streaming, compared to executing skeleton implementations using synchronous streaming, for different user functions, skeletons and number of SHAVE processors. User function 1 performs the least amount of calculations; 5 the most.

6.4 Speedup of Asynchronous DMA Streaming

Figure 6.11 shows the reduction in execution time from the use of asynchronous DMA streaming, for unary Map and Generate using different user functions. The user functions range in different number of calculations per vector element, where the user function labeled 1 performs the least amount of calculations, and 5 the most. A higher user function number means more calculations; for instance, user function 3 performs more calculations than user function 2. For each user function, skeleton, and number of SHAVE processors, an implementation of the corresponding skeleton using synchronous streaming is first run, followed by a run using an implementation of the same skeleton using asynchronous streaming. The problem size
remains constant for each skeleton execution. The time differences between the runs using asynchronous streaming and synchronous streaming are shown in Figure 6.11.

As described in Section 5.5, the speedup gained from asynchronous DMA streaming depends on the overlap between calculations and DMA transfers. By varying the number of calculations, the overlap between calculations and DMA transfers changes, leading to different speedups, as observed in Figure 6.11. If we for instance look at the Map skeleton running on 1 SHAVE processor, we see that the time reduction caused by asynchronous streaming is almost the same for user functions 3, 4, and 5; which indicates that the overlap between calculations and DMA transfers is the same. When the time spent on calculations is increased beyond the time spent on DMA transfers (by increasing the number of calculations in the user function), the overlap between calculations and DMA transfers remains constant.

6.5 Using Different DMA Buffer Sizes

As stated in Section 5.5, most skeletons stream operand data and results between the DDR memory and the CMX memory, using DMA. A buffer in the CMX memory is used to store chunks of operand data and results during calculations. A smaller buffer size will lead to more, shorter DMA transfers, and a large buffer size will lead to fewer, longer DMA transfers.

Figure 6.12 shows the execution times for three different applications, using different amount of SHAVE processors, and different DMA buffer sizes. In the figure, 1, 6, and 12 denotes the number of SHAVE processors used. MA denotes an application using the MapArray skeleton to reverse a vector of $10^5$ elements. G denotes an application using the Generate skeleton to calculate the Chebyshev (chessboard) distances in a 1024x1024 matrix. M denotes an application using the Map skeleton to calculate the square root on a vector of $10^5$ elements.

We can see that varying the DMA buffer sizes produces small differences in the execution times. A larger buffer size slightly improves performance in the tested applications when using only 1 SHAVE processor. In the applications using MapArray and Map, a smaller buffer size slightly improves performance when using all 12 SHAVEs.
6.6. SkePU Example Applications

The default buffer size used in the SkePU Myriad 2 back-end is 12 kB. Note that using a too large buffer size will reduce the available space for variables and constants in the skeleton code and user functions, which can cause the space in the CMX memory to run out \[^{[21]}\]. Using a too small buffer size can reduce performance in skeletons which require a minimum amount of buffer size in able to perform streaming. For instance, skeletons that apply user functions on whole matrix rows, need a buffer that can contain a whole row, and will fall back to accessing the DDR memory directly (which is slower as shown in figures 6.7 and 6.8) when a whole row cannot fit into the buffer.

6.6 SkePU Example Applications

SkePU is distributed with a number of example applications. The example applications range from more simple ones, using a single skeleton to solve a problem, to more complex ones, using several different skeletons to solve different parts of a problem.
A number of SkePU’s example applications were benchmarked on Myriad 2 using the Myriad 2 back-end [21]. Figure 6.13 shows the execution times for an example application performing LU decomposition on a 256x256 matrix for different number of SHAVE processors. Figure 6.14 shows the execution times for an example application performing median filtering on an image of 512x512 pixels, where each pixel consists of 3 bytes, for different number of SHAVE processors and filter window sizes. Figure 6.15 shows the execution times for an example application performing an SPH fluid dynamics for different number of SHAVE processors and simulated particles. We see that the back-end can run already existing SkePU applications, and the performance increases with the number of SHAVE processors used.

6.7 Energy Efficiency

The average power consumption of the Myriad 2 chip, and of a PC with a GPGPU, was measured in three different SkePU example applications [21]. The PC contained an Intel Xeon E5-2630L v2 CPU and an Nvidia Tesla K20c GPGPU. SkePU’s CUDA backend was used on the PC. For the Myriad 2, an MV0198 power measurement daughter card was used to measure the power consumption. For the PC system, the MeterPU [12] library was used to measure the aggregate power consumption of the CPU, DRAM, and GPGPU.

Table 6.1 shows the results from the measurements. Time, power, and efficiency are shown. Efficiency is calculated by \( \frac{1}{\text{power} \cdot \text{time}} \), where \( \text{power} \cdot \text{time} \) gives the power-delay product.

Table 6.1: Energy Efficiency Comparison Between PC and Myriad 2. © 2016 IEEE.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512 LU decomposition on PC</td>
<td>48.90</td>
<td>130.9</td>
<td>0.1562</td>
</tr>
<tr>
<td>512x512 LU decomposition on Myriad 2</td>
<td>4963</td>
<td>0.6899</td>
<td>0.3624</td>
</tr>
<tr>
<td>SPH on 1024 particles on PC</td>
<td>177.4</td>
<td>103.8</td>
<td>0.05431</td>
</tr>
<tr>
<td>SPH on 1024 particles on Myriad 2</td>
<td>843.5</td>
<td>0.6831</td>
<td>1.788</td>
</tr>
<tr>
<td>Dot product between two 10^7 long vectors on PC</td>
<td>0.8436</td>
<td>76.53</td>
<td>15.49</td>
</tr>
<tr>
<td>Dot product between two 10^7 long vectors on Myriad 2</td>
<td>32.68</td>
<td>1.047</td>
<td>29.23</td>
</tr>
</tbody>
</table>

Table 6.1: Energy Efficiency Comparison Between PC and Myriad 2. © 2016 IEEE.

The average power consumption of the Myriad 2 chip, and of a PC with a GPGPU, was measured in three different SkePU example applications [21]. The PC contained an Intel Xeon E5-2630L v2 CPU and an Nvidia Tesla K20c GPGPU. SkePU’s CUDA backend was used on the PC. For the Myriad 2, an MV0198 power measurement daughter card was used to measure the power consumption. For the PC system, the MeterPU [12] library was used to measure the aggregate power consumption of the CPU, DRAM, and GPGPU.

Table 6.1 shows the results from the measurements. Time, power, and efficiency are shown. Efficiency is calculated by \( \frac{1}{\text{power} \cdot \text{time}} \), where \( \text{power} \cdot \text{time} \) gives the power-delay product. All three tested SkePU applications runs slower, but use less energy, on the Myriad 2 than on the PC.
In this chapter we discuss the usefulness of our SkePU Myriad 2 back-end. We also draw conclusions based both on previous work on SkePU, and our work creating a SkePU back-end for Myriad 2. Finally, we propose 3 focus areas for future work.

7.1 Usefulness

Though our back-end does not reach the theoretical maximum performance for Myriad 2, the performance should be enough to consider using SkePU when programming applications targeting Myriad 2. As we have not compared any SkePU applications to any equivalent hand-written Myriad 2 applications, it is still unclear how far our back-end is from achieving the performance expected from Myriad 2. Doing a comparison between the performance of the back-end and the theoretical maximum performance of Myriad 2 can be an unfair comparison, as one does not necessarily reach maximum performance with a hand-written application. Nevertheless, the benchmarks we have performed in our evaluation of the back-end can give pointers on whether using SkePU for Myriad 2 is suitable for a particular case.

Our benchmarks have also shown that existing SkePU applications can run more energy-efficiently on Myriad 2 (with the Myriad 2 back-end) than on a GPGPU system. In cases where energy-efficiency is preferred over performance, one may want to choose to run an existing SkePU application on Myriad 2 instead of another back-end.

7.2 Conclusions

The SkePU back-end for Myriad 1 shows that it is possible for SkePU to support a heterogeneous multicore system-on-a-chip. Based upon this previous work, we have created a new and, performance-wise, improved SkePU back-end for Myriad 1’s successor: Myriad 2. We have argued for the usefulness of our new back-end, and shown that SkePU can achieve decent performance on Myriad 2. This in turn shows that SkePU, and algorithmic skeleton programming in general, is applicable to both supercomputer systems consisting of many processors and GPGPUs; and low-power embedded systems such as Myriad 2.
7.3 Future Work

Evaluation
In order to get a better understanding of the performance of our back-end, it is necessary to make comparisons between SkePU applications and real-world hand-written applications for Myriad 2. When making comparisons, it can also be useful to perform program profiling, to see where in the back-end overhead is produced.

DMA Data Streaming
We know from our evaluation that the DMA data streaming used in the back-end becomes a bottleneck when the amount of calculations done per data element is too small. The performance of the DMA data streaming can possibly be improved by redesigning how data streaming is performed during skeleton execution, as explained in Section 6.1. However, changing how the data streaming is performed will require touching many parts of the back-end; additional evaluation of different streaming techniques should arguably be performed before making such a change.

Hardware Filters
Myriad 2 contains a number of high-performing hardware filters, such as sharpen filter and median filter. A median filter application could, instead of using the SHAVE processors to perform the filter calculations, use the median filter that is included in the hardware, to achieve greater performance. For SkePU to utilise these hardware filters, user functions which correspond to available hardware filters would have to be identified before or during the compilation of SkePU applications.
List of Common Application Programming Interfaces for Different Parallel Computer Systems

- **CUDA**: Compute Unified Device Architecture. Used for programming applications for Nvidia GPGPUs.

- **MPI**: Message Passing Interface. Used for programming applications for computer clusters.

- **OpenCL**: Open Computing Language. Used for programming applications for GPGPUs, as well as digital signal processors, field-programmable gate arrays, and other applicable hardware.

- **OpenMP**: Open Multi-Processing. Used for programming applications for multicore and multi-processor systems.

- **pthreads**: POSIX Threads. Used for programming applications for multicore and multi-processor systems adhering to the POSIX operating system interface standard.
Bibliography


[17] moviCompile, Movidius Compiler 00.50.69.4. Movidius Ltd. 2015.


