Extracting Data Parallel Computations from Distilled Programs

Venkatesh Kannan and G. W. Hamilton

School of Computing, Dublin City University, Ireland
{vkannan, hamilton}@computing.dcu.ie

Abstract. To effectively utilise the parallel computing power of the heterogeneous architecture in hardware, potential parallelism in programs needs to be extracted and characterised. The extraction of parallel computations in a given program, though challenging and error-prone in practice, should be automated for both efficiency and accuracy of the parallelisation process.
In this paper, we present our initial work to automate the identification of data parallel computations in a given functional program for their execution on heterogeneous hardware with multi-core CPUs and GPUs. To achieve this, we use a program transformation technique called *distillation* [11, 12], and a data type transformation technique used in *AutoPar* [10] to transform an arbitrary program to operate over flat data types. We then choose a set of *skeletons* that are widely used for parallel program development [8,9,13,14], and use their characteristics to identify and extract instances of the skeletons from the transformed program that operates over flat data types. Following this, we replace these skeleton instances with their equivalent operations in the *Accelerate* library [4], which provides efficient OpenCL implementations for their execution on multi-core CPUs and GPUs.
We are presently working on formally specifying our parallelisation process, before comprehensively evaluating the parallel programs produced by our approach against expert hand-written parallel programs.

1 Introduction

The architecture of today's computing systems is made up of a heterogeneous collection of parallel processing units. The most common parallel processing units found are multi-core CPUs and many-core GPUs. CPUs are better suited to efficiently executing latency-critical programs that may have dynamic control-flow. GPUs, on the other hand, are designed for efficient execution of throughput-critical programs that have minimal control-flow divergence and a large number of identical threads.

In this setting, the development of parallel programs is vital to harness the computing power available in hardware. When it comes to parallelisation of programs and their execution, there are some tasks to be done either by the programmer, or by the implementor of the parallel programming system [8].

– *Problem decomposition*: Identification of computations in a program that can be executed in parallel.
– *Distribution*: A mapping from computations that may be executed in parallel to the available processing units.
– *Code and data sharing*: Decisions on how to spread the code and data for the computations to be executed in parallel across the chosen architecture, aiming at a performance improvement over a sequential execution.
– *Communication and synchronisation*: A mechanism that describes resource sharing and control.

Parallelism in a program can be *implicit* or *explicit* depending on which of the above tasks are specified by the programmer, and which are specified by the implementor in the programming system [8]. A completely explicit parallel program will have all four of the above tasks specified by the programmer, while a completely implicit parallel program will have all of them implemented in the parallel programming system.

Parallelisation of a given program, on the other hand, can be either *manual* or *automated*. In manual parallelisation, given the existing complexity of implementing an algorithm from its design, manually identifying and expressing parallel code can make development tedious and error-prone. Alternatively, automated program parallelisation involves identifying computations in a program that exhibit parallelism through program analysis. Such parallel computations can then be extracted and expressed explicitly using program transformation techniques. However, in practice, such automated parallelisation can be quite difficult for an arbitrary given program, especially while targeting its execution on a heterogeneous parallel architecture.

In this paper, we present our initial work that uses a program transformation technique called *distillation* [11,12], and extracts potential parallel computations from *distilled* programs. This automates the task of problem decomposition, thus making potential parallelism in a given program more obvious. For the purpose of this paper, a detailed description of distillation is not required; it is sufficient to know that the *distilled* expressions are in a specialised form called *distilled form*. To identify and extract parallel computations, we choose a set of *skeletons*, which are algorithmic forms that are common to a wide range of parallelisable problems [8]. The extracted parallel computations are then scheduled for execution on CPUs and GPUs based on their characteristics.

The remainder of this paper is structured as follows. In Section 2, we define the syntax and semantics of the higher-order functional language which we use in the parallelisation process. In Section 3, we elaborate on the characteristics of parallel computations that we use to decide their scheduling on a CPU or a GPU for execution. Also presented in this section are the functional definitions of the chosen skeletons to encompass these characteristics. In Section 4, we present our method to transform data types of a given program into a form that makes it amenable to parallelisation, and our parallelisation technique. In Section 5, we outline the course planned to complete this work. In Section 6, we summarise by considering related work in this context.

2 Language

In this work, we focus on program parallelisation applied to functional languages. This is primarily due to certain advantages that functional languages have. The lack of side-effects in pure functional languages is a major benefit, which makes them easier to analyse, reason about, and manipulate using program transformation techniques. The lack of side-effects also allows parallel evaluation of independent sub-expressions in a program. The higher-order functional language used in this work is presented in Definition 1.

Definition 1 (Language Syntax).

$$
\begin{array}{lll}
e ::= & x & \textit{Variable} \\
\mid & c\ e_1 \ldots e_k & \textit{Constructor Application} \\
\mid & \lambda x.e & \lambda{-}\textit{Abstraction} \\
\mid & f & \textit{Function Call} \\
\mid & e_0\ e_1 & \textit{Application} \\
\mid & \mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k & \textit{Case Expression} \\
\mid & \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 & \textit{Let Expression} \\
\mid & e_0\ \mathbf{where}\ f_1 = e_1, \ldots,\ f_n = e_n & \textit{Local Function Definitions} \\
 & & \\
p ::= & c\ x_1 \ldots x_k & \textit{Pattern}
\end{array}
$$

A program in this higher-order language is an expression e, which can be a variable, constructor application, $\lambda-$abstraction, function call, application, **case**, **let**, or **where**. Any variables introduced in the $\lambda-$abstraction, **case** patterns or **let** are *bound*, while all other variables are *free*. Each constructor has a fixed arity. In an expression $c\ e_1 \ldots e_k$, k must be equal to the arity of the constructor c. The patterns in **case** branches may not be nested. Techniques exist to transform nested patterns into equivalent non-nested versions [1,19]. No variable may appear more than once within a pattern and it is also assumed that all patterns are non-overlapping and exhaustive.

3 Parallel Computations and Skeletons

To be able to identify potential parallelism in a given functional program, we classify parallel computations into two categories - *data parallel* and *task parallel* computations. Since GPUs are capable of efficiently executing a large number of identical threads that have minimal control-flow divergence, data parallel computations are better suited for execution on GPUs. However, the cost of transferring data between the system main memory and the GPU memory is non-trivial. Hence, given enough computational work per unit data, data parallel computations that operate on significantly large datasets will have a larger performance gain when executed on GPUs. All other computations (sequential, data parallel computations operating on smaller datasets, and task parallel computations) are scheduled for execution on CPUs.

Data parallelism can be further classified as *flat* and *nested* data parallelism. Flat data parallelism executes more efficiently on GPUs, as opposed to nested data parallelism. This is because flat data parallelism is closer to the Single Program Multiple Data (SPMD) model that the GPU hardware is based on. Also, the control-flow is more regular and less divergent in the case of flat data parallelism allowing high throughput during execution. Hence, our objective is to

1. transform any given program to operate over flat data types, and
2. identify all potential flat data parallel computations in the transformed program.

We represent flat data parallel computations using *skeletons*, which are algorithmic forms that are common to a wide range of parallelisable problems. Our choice of skeletons is based on Blelloch's work on a vector-model for data parallel computations [2] that includes a study of primitive operations required to implement data parallel computations. To encompass the characteristics of data parallelism, we choose three *skeletons* – *map*, *reduce* and *zipWith*. These skeletons are also widely used in the development of programs that have data parallel computations [8, 9, 13, 14].

The three skeletons defined over lists are presented in Definition 2.

Definition 2 (Skeletons Defined over Lists).

```
$map\ f\ xs$
$\textbf{\textit{where}}$
$map = \lambda f.\lambda xs.\textbf{case}\ xs\ \textbf{of}$
                    $Nil \rightarrow Nil$
                    $Cons\ x'\ xs' \rightarrow Cons\ (f\ x')\ (map\ f\ xs')$

$reduce\ v\ f\ xs$
$\textbf{\textit{where}}$
$reduce = \lambda v.\lambda f.\lambda xs.\textbf{case}\ xs\ \textbf{of}$
                         $Nil \rightarrow v$
                         $Cons\ x'\ xs' \rightarrow reduce\ (f\ x'\ v)\ f\ xs'$

$zipWith\ f\ xs\ ys$
$\textbf{\textit{where}}$
$zipWith = \lambda f.\lambda xs.\lambda ys.$
               $\textbf{case}\ xs\ \textbf{of}$
                    $Nil \rightarrow Nil$
                    $Cons\ x'\ xs' \rightarrow \textbf{case}\ ys\ \textbf{of}$
                                          $Nil \rightarrow Nil$
                                          $Cons\ y'\ ys' \rightarrow Cons\ (f\ x'\ y')$
                                                         $(zipWith\ f\ xs'\ ys')$
```

- *map* : applies a function f to each element in a list xs, and produces a list of the same size as that of the input.
- *reduce* : collapses a list xs into a single value using an associative binary operator f, with a unit value v, by accumulating the reduction value.
- *zipWith* : combines two lists xs and ys point-wise using a binary operator f on the corresponding elements.

4 Program Parallelisation

Before we apply our parallelisation technique to identify potential data parallel computations in a program, we need to identify if the program may contain data parallelism. In this context, we observe that only some of the data types over which a program is defined are suited to data parallelism such as lists, trees or arrays. Hence, we allow the developer to specify a set of *parallelisable types*, γ, to which our parallelisation technique can be applied.

In our parallelisation approach, we identify instances of the three list skeletons in the result of applying distillation to a given program. To facilitate this, the parallelisable types within a program need to be converted to flat lists. Hence, it is necessary to transform the program using distillation to operate over these flat lists in order to identify potential flat data parallel computations using our skeletons. This is explained in Section 4.1.

Upon transformation of the original program f to operate over lists, we obtain the transformed program f_{list}. Following this, we use the characteristics of the skeletons that are presented in Section 3 to identify and extract their instances from f_{list}. This is explained in Section 4.2.

Our approach to execute the identified skeletons efficiently on CPU or GPU is explained in Section 4.4.

4.1 Data Type Transformation

We use two sets of functions to transform a program f with input data of type T_{in} and output data of type T_{out}, into a semantically equivalent one, f_{list}, defined on list data types.

For input type $T_{in} \in \gamma$, and output type $T_{out} \in \gamma$,

- $flatten_{in}$ and $flatten_{out}$: these functions transform T_{in} and T_{out} into lists of their component types T'_{in} and T'_{out}.
- $unflatten_{in}$ and $unflatten_{out}$: these functions transform lists of component types T'_{in} and T'_{out} back to their corresponding original data types T_{in} and T_{out}.

The type signatures of these functions are presented in Definition 3. These functions use Dever's work on data partitioning [10] that defines functions $flatten$ and $unflatten$ to provide transformations between an instance of type T and a list of its component types T'.

Definition 3 (Signatures of Type Transformation Functions).

$$\begin{aligned} flatten_{in} &:: T_{in} \rightarrow (List\ T'_{in}) \\ unflatten_{in} &:: (List\ T'_{in}) \rightarrow T_{in} \\[1em] flatten_{out} &:: T_{out} \rightarrow (List\ T'_{out}) \\ unflatten_{out} &:: (List\ T'_{out}) \rightarrow T_{out} \end{aligned}$$

It is to be noted that if $T_{out} \in \gamma$, as in the case of *map* and *zipWith* skeletons, then $flatten_{out}$ transforms T_{out} into $List\ T'_{out}$. If $T_{out} \notin \gamma$, as in the case of *reduce* skeleton, then the output of $flatten_{out}$ is the original output data type T_{out}. The function $unflatten_{out}$ also works in a similar fashion.

As illustrated in Fig. 1, a composition of $unflatten_{in}$, the original program f, and $flatten_{out}$ yields a program that is defined over list data types.

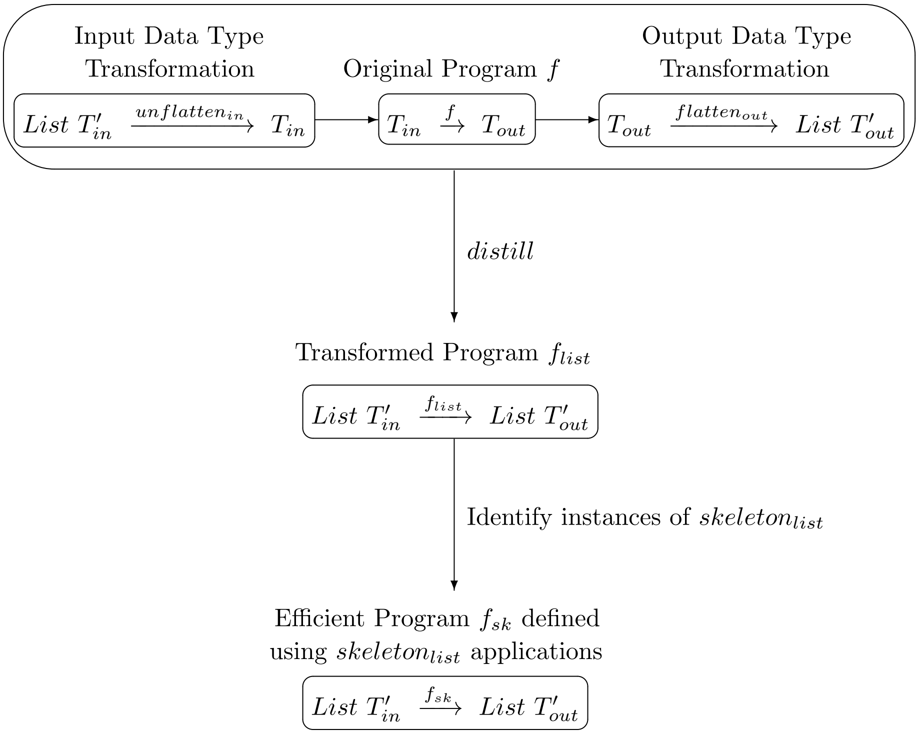
Input Data Type Transformation
Original Program f
Output Data Type Transformation
$List\ T'_{in} \xrightarrow{unflatten_{in}} T_{in}$
$T_{in} \xrightarrow{f} T_{out}$
$T_{out} \xrightarrow{flatten_{out}} List\ T'_{out}$
$distill$
Transformed Program f_{list}
$List\ T'_{in} \xrightarrow{f_{list}} List\ T'_{out}$
Identify instances of $skeleton_{list}$
Efficient Program f_{sk} defined using $skeleton_{list}$ applications
$List\ T'_{in} \xrightarrow{f_{sk}} List\ T'_{out}$

Fig. 1. Transformation of Original Program

4.2 Parallelisation Technique

Fig. 1 also illustrates our parallelisation technique. As a first step, we *distill* the composition of $unflatten_{in}$, f and $flatten_{out}$. This yields a program f_{list}, which is semantically equivalent to f and is defined on list data types.

The definition of f_{list} is in the *distilled form*, which is presented in Definition 4. In an expression in the distilled form, de^{ρ}, resulting from distillation, ρ denotes the set of variables that have been introduced in **let** expressions, and cannot therefore appear as a selector in a **case** expression. As a result of this, expressions in distilled form do not create intermediate data structures.

Definition 4 (Syntax of Distilled Form).

$$
\begin{array}{lll}
de^{\rho} ::= & x & \\
| & c\ de_1^{\rho} \ldots de_k^{\rho} & \\
| & \lambda x . de^{\rho} & \\
| & f & \\
| & de^{\rho}\ x & \\
| & \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow de_1^{\rho} \mid \ldots \mid p_k \rightarrow de_k^{\rho} & \qquad \textbf{\textit{where}}\ x \notin \rho \\
| & \textbf{let}\ x = de_0^{\rho}\ \textbf{in}\ de_1^{(\rho\, \cup\, \{x\})} & \\
| & f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots\ \lambda x_n . de^{\rho} &
\end{array}
$$

Additionally, we have found that the three skeletons described in Section 3 can be associated with the following three characteristics of recursive functions that a given program in distilled form may have.

- *Case 1* : A recursive function has one decreasing parameter, which is of the same type as the result. This would indicate the presence of a *map*-like computation.
- *Case 2* : A recursive function has one decreasing parameter, which is of a different type to the result. This would indicate the presence of a *reduce*-like computation.
- *Case 3* : A recursive function has more than one decreasing parameter. This would indicate the presence of a $zipWith$-like computation.

In addition to problems that fit one of the three cases mentioned above, we may also have problems with any combination of these cases indicating the need for a composition of skeletons.

Using these characteristics of sub-expressions, we identify instances of the skeletons in the transformed program f_{list}. As a result, instances of the skeletons embedded in f_{list} are extracted. This yields the program f_{sk} that is defined using applications of the skeletons $skeleton_{list}$.

4.3 An Example : Find Maximum

The parallelisation of a program, $findMax$, to find the largest positive element in a list using the proposed parallelisation approach, is presented in Example 1.

The input program to the transformation process is shown in expression (1). Here xs is the list to be parsed through to find the largest element. The definition consists of two functions: *bigger* to find the larger of two given elements, and $findMax$ to find the largest positive element in a given list.

The distillation of expression (1), without identifying instances of list skeletons, produces the distilled form of $findMax$ presented in expression (2).

Expression (3) is the result of identifying list skeletons in expression (2) using our parallelisation technique. Here, f is the reduction operation that distillation has extracted in a definition that uses an application of the *reduce* skeleton.

Example 1 (Find Maximum in List).

Expression (1) : Original Program

$$\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow v \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow bigger\ x'\ (findMax\ xs'\ v) \\
bigger \quad\ = \lambda x.\lambda v.\textbf{case}\ (x > v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad True \qquad\ \rightarrow x \\
\qquad\qquad\qquad\qquad False \qquad \rightarrow v
\end{array}$$

Expression (2) : Distilled Program

$$\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow v \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow \textbf{let}\ v' \ =\ (\textbf{case}\ (x' > v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad True \ \rightarrow\ x' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad False \rightarrow\ v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ findMax\ xs'\ v'
\end{array}$$

Expression (3) : Distilled Program with Skeletons Identified

$$\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{let}\ f\ =\ \lambda x'.\lambda v'.(\textbf{case}\ (x' > v')\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \ \rightarrow\ x' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow\ v') \\
\qquad\qquad\qquad\quad \textbf{in}\ \ reduce\ v\ f\ xs
\end{array}$$

For parallel evaluation of the application of *reduce* skeleton, it is required that the reduction operator be associative. As a result, we have to prove the associativity of a reduction operator, f, used by the extracted *reduce* skeleton. This can be achieved by distilling the two expressions $f\ (f\ x\ y)\ z$ and $f\ x\ (f\ y\ z)$ using the definition of f. If the distilled forms of both expressions are syntactically equal, then f is associative.

4.4 Execution of Data Parallel Computations

The skeleton applications identified in the distilled program represent data parallel computations. Skeleton applications that work on smaller datasets are scheduled for execution on CPU, while those that are computation intensive and work on significantly larger datasets are scheduled for execution on GPU. This is due to the potentially larger overhead involved in shipping the data between the system main memory and the GPU memory. To allow the execution of the skeleton applications on CPUs and GPUs alike, we make use of the *Accelerate* library of operations [4].

Accelerate Library. This is a domain-specific purely functional high-level language embedded in Haskell. The library contains efficient data parallel implementations for many operations including the chosen skeletons : *map*, *reduce* and *zipWith*. We replace the identified skeleton applications with calls to the corresponding *Accelerate* library operations, which have efficient OpenCL implementations. This allows their scheduling and execution on CPUs and GPUs, among other OpenCL-compatible processing units.

The *Accelerate* library operations are defined over their custom *Array sh e* data type. Here, *sh* is a type variable that represents the shape of the *Accelerate* array. It is implemented as a heterogeneous *snoc*-list where each element in the list is an integer to denote the size of that dimension. A scalar valued array is represented in *sh* by Z, which acts as both the type and value constructor. A dimension can be added to the array by appending the size of that dimension to *sh*. The type variable e represents the data type of the elements stored in the *Accelerate* array.

To execute the data parallel computations in f_{sk} on a CPU or GPU, we replace each application of $skeleton_{list}$ with a call to the corresponding *Accelerate* library operation $skeleton_{acc}$. The resulting $skeleton_{acc}$ calls operate over the *Accelerate* array types; inputs of type $Array\ sh_{in}\ T_{in}^{sk'}$, and output of type $Array\ sh_{out}\ T_{out}^{sk'}$. Consequently, we need to transform the input and output data of the original program f to and from the *Accelerate* array type.

These transformations are illustrated in Fig. 2 and Fig. 3 as "Input Data Transformation" and "Output Data Transformation", and are explained below:

1. *Input Data Transformation*
   - $flatten_{in}$: This function transforms the input data for f of type $T_{in} \in \gamma$ into $List\ T_{in}'$ for input to f_{sk}.
   - Each skeleton application $skeleton_{list}$ in f_{sk} operates over input data of type T_{in}^{sk}. We replace these $skeleton_{list}$ applications with calls to corresponding $skeleton_{acc}$ operations that operate over *Accelerate* array types.

  - *toAcc* : This function transforms the input data for $skeleton_{list}$ of type T_{in}^{sk} into an *Accelerate* array of type $Array\ sh_{in}\ T_{in}^{sk'}$.
    To define *toAcc*, we use the $fromList$ function that is available in the *Accelerate* library [3], which creates an *Accelerate* array from a list.

2. *Output Data Transformation*
  - $fromAcc$: This function transforms the output from a $skeleton_{acc}$ operation of type $Array\ sh_{out}\ T_{out}^{sk'}$ back to the output type T_{out}^{sk} of the corresponding $skeleton_{list}$ application.
    To define $fromAcc$, we make use of the *toList* function that is available in the *Accelerate* library, which converts an *Accelerate* array into a list.
  - This output from the $skeleton_{list}$ application is then plugged back into its context in f_{sk}.
  - $unflatten_{out}$: This function transforms the output from f_{sk} of type $List\ T'_{out}$ back to the output of type T_{out} of the original program f.

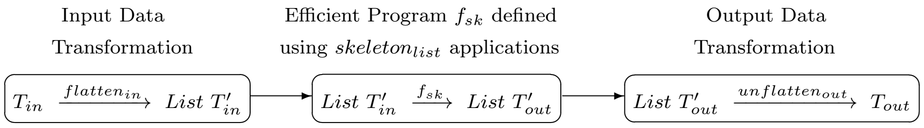
Input Data Transformation
$T_{in} \xrightarrow{flatten_{in}} List\ T'_{in}$
Efficient Program f_{sk} defined using $skeleton_{list}$ applications
$List\ T'_{in} \xrightarrow{f_{sk}} List\ T'_{out}$
Output Data Transformation
$List\ T'_{out} \xrightarrow{unflatten_{out}} T_{out}$

Fig. 2. Transformation of Data for Transformed Program f_{sk}

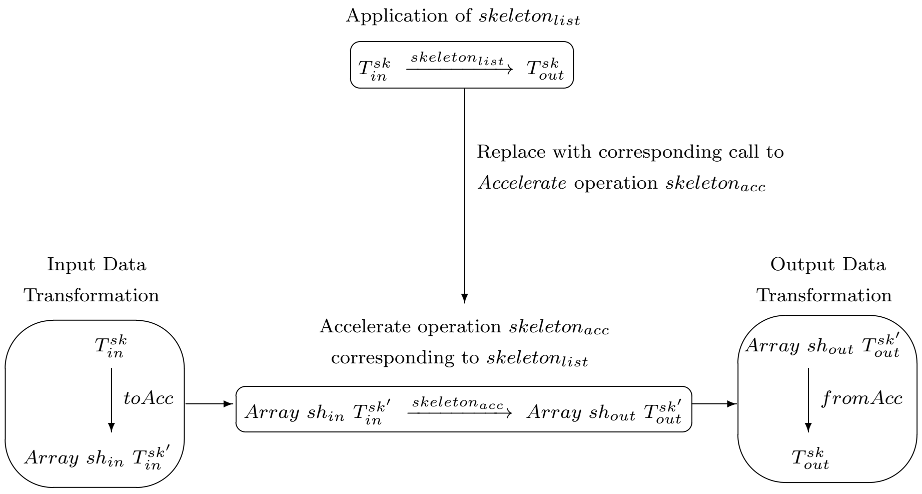
Application of $skeleton_{list}$
$T_{in}^{sk} \xrightarrow{skeleton_{list}} T_{out}^{sk}$
Replace with corresponding call to *Accelerate* operation $skeleton_{acc}$
Input Data Transformation
T_{in}^{sk}
$toAcc$
$Array\ sh_{in}\ T_{in}^{sk'}$
Accelerate operation $skeleton_{acc}$ corresponding to $skeleton_{list}$
$Array\ sh_{in}\ T_{in}^{sk'} \xrightarrow{skeleton_{acc}} Array\ sh_{out}\ T_{out}^{sk'}$
Output Data Transformation
$Array\ sh_{out}\ T_{out}^{sk'}$
$fromAcc$
T_{out}^{sk}

Fig. 3. Transformation of Data for *Accelerate* Operations $skeleton_{acc}$

5 Future Work

We are currently working on formally specifying the transformations rules to parallelise a distilled program, which was described in Section 4. Following this, we will comprehensively evaluate our approach qualitatively and quantitatively.

To evaluate our parallelisation approach, we require a suite of benchmark programs with diverse definitions for each. These programs will include vector dot-product, point-wise vector/matrix arithmetic, matrix multiplication, search algorithm, sort algorithm, histogram generation, image rotation, image convolution, string reverse algorithm, and maximum segment sum algorithm.

One part of our evaluation will be a qualitative analysis of the *coverability* of our representation of parallel computations and of the proposed parallelisation technique. We define *coverability* as the ability to identify potential data parallel computations in a spectrum of diverse definitions of benchmark programs. All the programs we have chosen for benchmarking have potential data parallelism. By distilling these programs and applying our transformation to identify the skeletons, we evaluate their coverability over different forms in which data parallel computations may be expressed in the benchmark programs. Following this, we will address the inclusion of additional skeletons or combinations of skeletons to identify data parallel computations that are otherwise manually identifiable.

Another part of our evaluation will be a quantitative analysis of the performance of our parallelisation technique and the execution of the parallelised programs that are produced. We will perform a quantitative analysis using the benchmark programs by comparing the performance metrics listed below for different configurations of CPU-GPU based hardware and OpenCL program execution environment settings such as varying sizes of datasets, number of threads created, number of work-groups, and work-group sizes. We will also determine the overhead involved in program and data transformation as a factor of the difference in execution times of parallelised and non-parallelised versions of the benchmark programs.

The performance metrics for the quantitative analysis are:

1. *Execution time*
   (a) Time to transfer data between system main memory and GPU memory.
   (b) Time to transfer code from system main memory to GPU memory.
   (c) Time to execute the identified data parallel computations on GPU, and remaining computations on CPU.
   (d) Time to execute the parallelised input program on a multi-core CPU.
   (e) Time to execute the given input program without parallelisation on CPU.
2. *Program transformation time*
   (a) Time to parallelise a given input benchmark program using our approach.
3. *Data transformation time*
   (a) Time to $flatten$ input data into lists.
   (b) Time to transform input data into *Accelerate* array using $fromList$.
   (c) Time to transform output data from *Accelerate* operations using *toList*.
   (d) Time to $unflatten$ the output data to get the final result.

With respect to OpenCL code, metrics 1a, 1b and 1c listed above will be collected by time-stamping the corresponding OpenCL APIs that are used to transfer code and data between the system main memory and the GPU memory, and to schedule the execution of OpenCL kernel functions on the device.

To collect the metrics 1c, 1d, 1e, 2a, 3a, 3b, 3c and 3d that are related to the execution of computations on CPU, we intend to use the *Criterion* [17] or the *ThreadScope* [18] performance measurement packages for Haskell.

6 Conclusion and Related Work

To summarise, we automate the process of extracting potential parallelism in a given functional program to enable its execution on a heterogeneous architecture with CPUs and GPUs. For this, we choose a set of skeletons that are widely used to define data parallel computations in programs. Presently, we aim to use the characteristics of these skeletons to identify and extract their instances from programs in distilled form. Applications of the identified skeleton instances will then be replaced with calls to the corresponding operators in the *Accelerate* library, which provides efficient implementations for the skeletons in OpenCL so that they can be executed on multi-core CPUs and GPUs. In the next steps, we will complete the transformations required to call *Accelerate* library operators, and evaluate the efficiency of the enlisted skeletons to identify potential parallelism in a suite of benchmark programs. This will also include evaluation of the speedups gained from executing the data parallel computations on the GPU. Finally, we plan to investigate enlisting additional skeletons to encompass more parallel computation patterns including task parallelism, and address a wider range of input programs.

Previously, the use of skeletons as building blocks during program development has been widely studied. The early works by Murray Cole [8] and Darlington et. al. [9] exhaustively investigate the use of higher-order skeletons for the development of parallel programs. They present a repertoire of skeletons that cover both task parallel and data parallel computations that may be required to implement algorithms. Later on, addressing the possible difficulties in choosing appropriate skeletons for a given algorithm, Hu et. al. proposed a transformation called *diffusion* in [13]. Diffusion is capable of decomposing recursive definitions of a certain form into several functions, each of which can be described by a skeleton. They also present an algorithm that can transform a wider range of programs to a form that is decomposable by diffusion. This work has further led to the *accumulate* skeleton [14], a more general parallel skeleton to address a wider range of parallelisable problems.

Following the use of skeletons as building blocks in parallel program development, there has been significant work on including parallel primitives as an embedded language in widely used functional languages such as Haskell. One of the earliest works can be traced to Jouret [16], who proposed an extension to functional languages with data parallel primitives and a compilation strategy onto an abstract SIMD (Single Instruction Multiple Data) architecture. Obsidian is a language for data parallel programming embedded in Haskell, developed by Claessen et. al. [6]. Obsidian provides combinators in the language to express parallel computations on arrays, for which equivalent C code is generated for execution on GPUs. An evaluation by Alex Cole et. al. [7] finds that the performance results from generating GPU code from Haskell with Obsidian are acceptably comparable to expert hand-written GPU code for a wide range of problems. Among others, *Accelerate* [4] provides a domain-specific high-level language that works on a custom array data type, embeddeded in Haskell. Using a library of array operations, which have efficient parameterised CUDA and OpenCL implementations, *Accelerate* allows developers to write data parallel programs using the skeleton-based approach that can be executed on GPUs.

Despite the extensive work on identifying and developing skeletons, these approaches require manual analysis and identification of potential parallelism in a problem during development. As stated earlier, this can be quite tedious in non-trivial problems. On the other hand, a majority of the literature on skeletons involve *map* and *reduce* for data parallel computations, which are integral in our work to automate parallelisation. We include the $zipWith$ skeleton to address problems that operate on multiple flat datasets.

Another option for a parallel programmer is Data Parallel Haskell (DPH), an extension to the Glasgow Haskell Compiler (GHC), which supports nested data parallelism with focus on multi-core CPUs. Though flat data parallelism is well understood and supported, and better suited for GPU hardware, nested data parallelism can address a wider range of problems with irregular parallel computations (such as divide-and-conquer algorithms) and irregular data structures (such as sparse matrices and tree structures). To resolve this, DPH, which focuses on such irregular data parallelism, has two major components. One is a vectorisation transformation that converts nested data parallelism expressed by the programmer, using the DPH library, into flat data parallelism [15]. The second component is a generic DPH library that maps flat data parallelism to GHC's multi-threaded multi-core support [5]. It is worth pointing out that our method to automate parallelisation includes flattening steps that are similar to the vectorisation transformation in DPH. This flattening step provides a common form to the transformed input program and our enlisted skeletons, thereby aiding in the extraction of flat data parallel computations.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (`www.lero.ie`).

References

[1] L. Augustsson. Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*, 1985.

[2] Guy E. Blelloch. Vector Models for Data-Parallel Computing. *The MIT Press*, 1990.

[3] Manuel M. T. Chakravarty, Robert Clifton-Everest, Gabriele Keller, Sean Lee, Ben Lever, Trevor L. McDonell, Ryan Newtown, and Sean Seefried. An Embedded Language For Accelerated Array Computations. *http://hackage.haskell.org/package/accelerate*.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, 2011.

[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. *Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, 2007.

[6] Koen Claessen, Mary Sheeran, and Joel Svensson. Obsidian: GPU Programming In Haskell. *Proceedings of 20th International Symposium on the Implementation and Application of Functional Languages (IFL 08)*, 2008.

[7] Alex Cole, Alistair A. McEwan, and Geoffrey Mainland. Beauty And The Beast: Exploiting GPUs In Haskell. *Communicating Process Architectures*, 2012.

[8] Murray Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. *MIT Press, Cambridge, MA, USA*, 1991.

[9] John Darlington, A. J. Field, Peter G. Harrison, Paul Kelly, D. W. N. Sharp, Qiang Wu, and R. Lyndon While. Parallel Programming Using Skeleton Functions. *Lecture Notes in Computer Science, 5th International PARLE Conference on Parallel Architectures and Languages Europe*, 1993.

[10] Michael Dever and G. W. Hamilton. Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs. *PSI'14, 8th International Andrei Ershov Memorial Conference*, 2014.

[11] G. W. Hamilton. Distillation: Extracting the essence of programs. *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2007.

[12] G. W. Hamilton and Neil D. Jones. Distillation With Labelled Transition Systems. *Proceedings of the ACM SIGPLAN 2012 workshop on Partial Evaluation and Program Manipulation*, 2012.

[13] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating Efficient Parallel Programs. *In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 99)*, 1999.

[14] Hideya Iwasaki and Zhenjiang Hu. A New Parallel Skeleton For General Accumulative Computations. *International Journal of Parallel Programming*, 2004.

[15] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, 2008.

[16] Guido K. Jouret. Compiling Functional Languages For SIMD Architectures. *Parallel and Distributed Processing, IEEE Symposium on*, 1991.

[17] B. O'Sullivan. The Criterion Package. *http://hackage.haskell.org/package/criterion*.

[18] Satnam Singh, Simon Marlow, Donnie Jones, Duncan Coutts, Mikolaj Konarski, Nicolas Wu, and Eric Kow. The ThreadScope Package. *http://hackage.haskell.org/package/threadscope*.

[19] Philip Wadler. Efficient Compilation of Patten Matching. *S. P. Jones, editor, The Implementation of Functional Programming Languages*, 1987.