Universiteit Utrecht

Faculteit Bètawetenschappen

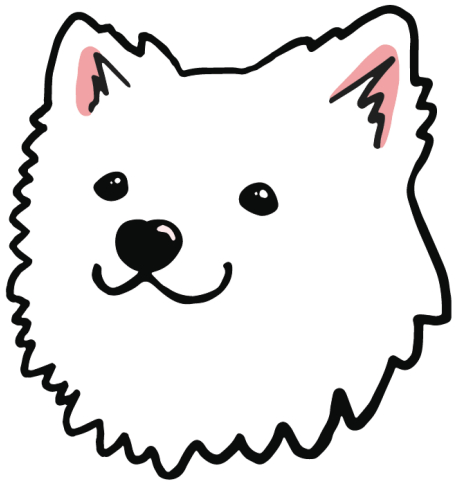
Automatic task parallelism in Accelerate

Master Thesis

Marien Matser BSc

6384013

Computing science



Supervisors:

Ivo Gabe de Wolff MSc
Information and computing science

Prof. Dr. Gabriele Keller
Information and computing science

Dr. Trevor McDonell
Information and computing science

Dr. Wouter Swierstra
Information and computing science

July 25, 2023

Abstract

Parallel execution has a large potential to improve the speed of a program. To make good use of parallelism we need a program that is properly split into several tasks that can be performed in parallel. High level frameworks like Accelerate make it easier to create such programs. Accelerate creates programs using data parallel array computations. For more parallelism we propose a way to automatically add task parallelism to programs as part of the Accelerate compiler. We define a transformation to introduce forks to a program. We formalize this transformation using inference rules and we implement it in the Accelerate compiler. To reduce the overhead of the added task parallelism, we propose several optimizations to not introduce forks where they do not introduce actual opportunities of added parallelism, like not forking trivial statements and combining statements that directly wait on the result of the previous statement. Our results show that the compilation of this implementation is not significantly slower than the current Accelerate compiler.

Contents

1 Introduction

Current hardware is often designed to be able to run multiple parts of a program in parallel, either on a CPU with multiple cores or massively parallel on a GPU. Since it can be difficult to manually create programs that use parallelism, tools and frameworks have been designed to make it easier for developers to develop parallel programs, like Accelerate[1], Accelerator[2], GPU++[3], Nikola[4], PyCUDA[5] and Sh[6].

1.1 Parallelization

There are multiple ways to have parallelism in programs. One of these ways is task parallelism, where we perform multiple different tasks of the program in parallel. Another way is data parallelism. Here we do not parallelism the different tasks, but we perform the same task on multiple parts of the data in parallel. These methods can also be combined to have different tasks being performed at the same time, each being performed on multiple parts of the data.
It is possible to manually add these types of parallelism to a program. It is, however, a lot easier if this is done automatically. Automatic parallelization has proven to be difficult for the general case[7–10]. This is partly due to the fact that we need some sort of way to determine when to stop parallelizing, because if we create too small tasks the parallel overhead is more than the speedup gained by parallelization. In Accelerate however the primitives that we work with are very large, as they are operations on entire arrays. This means that this is not as big of an issue for our case.

1.2 Accelerate

Accelerate is a domain specific language embedded in Haskell to write programs for working with large arrays, which can be executed in parallel on the GPU or the CPU. Currently Accelerate is mainly focused on data parallelism. We propose a way to add automatic task parallelism to the Accelerate compiler. The Accelerate compiler should automatically determine which operations in the code can be executed in parallel and create the structure so this can be done. This could lead to even more parallelism than the current Accelerate versions, so we can make more efficient use of the GPU or CPU.
Automatically adding task parallelism to Accelerate results in a quite complicated program transformation. We, therefore, want some assurances that our transformation is correct. For this we make a separate, slightly simplified, theoretical definition of the transformation, before implementing it in the code. This makes the transformation easier to understand. Additionally we want the types that are part of the Accelerate language and compiler to be kept during the transformation, so some incorrect programs are impossible to be created.

1.3 Research questions

The goal of this thesis is to add automatic task parallelism to Accelerate in a well defined and efficient way. This is done as one of the passes in the compiler. For this we present the main research question:

Research Question 1. *How can we add automatic task parallelism to Accelerate at compile time in a well defined, type safe and efficient way?*

To elaborate on this question we introduce the following sub research questions:

Research Question 2. *How can the program transformation be formalized?*

Research Question 3. *How can we ensure type safety?*

Research Question 4. *How much speedup can we gain with the added parallelism in executing Accelerate programs?*

2 Background

To understand how we implement automatic task parallelism and why, we first need some extra background information. In this section we discuss some previous work on parallelization, Accelerate and program transformations.

2.1 Parallelization

Parallelization is a common topic of research[7, 11–13]. Running code in parallel is often a good way to speed up a program. It can, however, be complicated to do this efficiently. We first need to be able to create a program that can be run in parallel. Which means we need multiple parts of the program that do not depend on each other, so we can execute them in parallel. After this we also need to be able to execute these parallel programs efficiently.

2.1.1 Parallel code

To execute a parallel program we first need a parallel program. For this, we need a way to represent parallelism in a language. There are multiple methods that can be used for this. A simple way is using fork-join. This works by having fork statements in the language that represent a split in the program path where the two sub-statements can be executed in parallel. After these statements are finished executing, they are joined again into a single thread. An example of this for a lazy functional language is the `letpar` expression, as used by Hogen et al. [9]. This works like a regular `let` expression, except for the fact that the expression that is bound in the `let` can be computed in parallel. To make sure the laziness is still satisfied, the `letpar` expression has an extra denotation for what level the parent expression need to be evaluated to, to make sure this binding is needed.

2.1.2 Automatic parallelization

The easiest way for a developer to create a parallel program is if this is done automatically. That way the developer can just write a sequential program and the compiler automatically changes this to a program that can be executed in parallel. This is, however, not that easy to implement. There have been multiple attempts to do this[7–10].

An early attempt was done by Burke et al. [8]. In this attempt they designed the parallelization for a simplified language to make it easier. The target parallel language that is used is a language that has two ways of parallel execution. One way is with loops for which the iterations can be executed in parallel. The other way is a construct that can be used to specify that multiple code blocks can be executed in parallel. The algorithm for automatically parallelizing a program to this target language works in three steps. The first step is the initialization step. In this step, all loops are initialized to parallel loops. Additionally, all statements that are executed with the same control flow are put to be in parallel. After this step there are still data dependencies that can cause this overly parallelized program to not always execute correctly. To fix this the second step loops over all data dependencies. For each data dependency we first find out whether the data dependency can be solved using privatization. Privatization works by creating a private, thread-specific instance of shared variables, so the process can just use this variable without having a problem when other processes also use it. Privatization can resolve dependencies because sometimes multiple parts of the program use the same variable, but do not actually depend on the value calculated in another part. If the dependency cannot be solved using privatization, the dependency is solved using sequencing. This means that the parallelism is reduced to make sure the data dependency is satisfied. In the third and final step, the privatizations found in the first part of the second step are actually added to the program. This is done at this point in the algorithm because some data dependencies that could be fixed by privatization are already fixed using sequencing, so we do not add the privatizations for these dependencies.

This first attempt works for a simplified imperative language. If the language has laziness, however, it becomes a lot harder. This is because certain parts of the code might not be executed, as the results are not actually needed. We cannot just parallelize everything that could be parallelized in terms of data dependencies, because we might be executing things that do not need to be executed. Hogen et al. [9] made an attempt at automatically parallelizing a lazy functional language. Their method works in three steps. The first step is λ-Lifting. This works by extracting nested function definitions to the global scope, to make sure they are available everywhere. The next step is strictness analysis. In this step, we annotate all expressions with their strictness properties. This is how far this expression will definitely be evaluated if the parent expression is evaluated to a certain level. We, thus, know when the result is actually used and can therefore be calculated beforehand or in parallel. This makes sure that we do not execute statements that are not actually needed. The final step is the parallelization. Here the expressions that can be parallelized according to the previous step are looked at. A heuristic is then used to determine whether parallelizing the expression can actually yield an improvement. If it is determined that it yields an improvement it is abstracted to a `letpar` expression.

2.1.3 Work stealing

When executing a program that has parallelism implemented, we want the work to be properly spread over the available resources. We need a strategy to distribute the work over the threads. If the work is not balanced properly it might be that one thread is still doing almost all the work while the other threads are idling. One strategy we can use for this is work stealing[14]. For work stealing each thread will have a queue with tasks that can be preformed in parallel. When a thread encounters part of a program that can be performed in parallel, like a fork, it can spawn a new task, which it puts in its queue. When another thread runs out of work it can look at the queues from the other threads and "steal" a task. This thread can then perform the "stolen" task. To make sure this can be done efficiently the queue for each thread can be implemented with a double ended circular queue [15].

2.2 Accelerate

Accelerate is an embedded domain specific language for general parallel programming. It is implemented in Haskell, and is able to generate optimised CUDA code [1] or efficient parallel code for the CPU. We can, for example, write the dot product of two vectors as

```
dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in
             fold (+) 0 (zipWith (*) xs' ys')
```

Here, the Acc in the type indicates that it represents a calculation that can later be executed on the device. The `fold` and `zipWith` are functions from Accelerate that perform the same operation as their counterparts from standard Haskell, but parallelized. The `use` function is used to lift the vectors from the CPU host memory to the device memory.
This idea however still comes with some difficulties. Because Accelerate compiles high level Haskell code to lower level parallel code, it needs to do some extra optimizations and program transformations to generate efficient parallel code. One of these optimizations will be the addition of task parallelism in this research, but there have already been other optimizations and improvements. Some of these are discussed in this section.

2.2.1 Shared code

Due to the embedding of Accelerate in Haskell, shared let bindings are lost. This means that if the expression bound in a let binding is used multiple times, it is reevaluated every time. Sharing recovery [16] can be used as an optimization. This is done by first discovering shared subterms by giving all terms a unique name, which is the same if and only if the terms are the same. If we then encounter a term we have already found, we replace it with a placeholder, since this is a shared term. Next these shared terms are floated to the top of the AST. Finally we can introduce a new binding for the terms, so they can be evaluated a single time and used in the places of their corresponding placeholders.

2.2.2 Fusion

When manually writing CUDA code, we would combine multiple operations on the same array by putting them in a single loop, to not create intermediate arrays. Because Accelerate is a combinator language where we use complete array operations, we just use multiple of these array operations in a row. A naive implementation, however, executes these operations after each other, making an intermediate array between each operation. This results in a lot of extra memory usage. To improve this, certain operations are performed at the same time using fusion [16]. For example, two map operations can be performed by simply applying both functions to each element: `map` f (`map` g xs) = `map` (f . g) xs.
To implement this, we can categorize the operations supported by Accelerate into two categories. The first one is for operations where each output element depends on at most one input element, called producers, like maps. The second one is for operations where each output element can depend on multiple input elements called consumers, like folds. We can relatively easily implement producer/producer fusion as we can just apply both producer functions after each other for each output element. If we have a producer and then a consumer we can also fuse this since the consumer can apply the function from the producer and apply it to the needed elements.

2.2.3 Foreign function interface

For some algorithms, there already exists highly optimized CUDA code. This code can often be a lot faster than the code generated by Accelerate. Therefore it is useful to be able to use this code from within an Accelerate program to execute parts of the total program [17].
To make the foreign code available we can use the regular foreign function interface in Haskell. We then need to lift this code to be able to run it on the device we want to run the code on. To then be able to call the foreign function from the Accelerate code we can add an extra node type to the AST. This node represents the foreign function to be executed. This node can however not just represent the foreign code, because Accelerate can compile to multiple different backends and the foreign code might not work with all of these backends. We therefore also need and extra backup function to execute when the foreign function can not be executed from the currently used backend.

2.2.4 Type safe code generation

Accelerate is an embedded language that is compiled at the runtime of the host program. Because of this a compile error in the Accelerate compiler results in a runtime error in the host program. If we make sure the compile pipeline is type safe [18], these problems occur a lot less. Because, when compiling the host program we can already guarantee that the types of the program are correct. For this, Accelerate uses GADTs to represent the typed AST of the Accelerate program within Haskell at each step of the compile pipeline. To represent what types can be used as array elements and in scalar expression within the AST, we use the Elt type class.
Because we want the compiler to be type-safe, we can add separate optimizations in terms of type safe program transformations. These optimizations can be done separately from other optimizations, which makes the compiler just a series of optimizations and compilations which all preserve the types in the program. We can, for example, use this to implement array fusion and other previous optimizations again in a type safe manner.
In the end, the code is compiled to LLVM's intermediate language. This, however, does not have proper type safety guarantees. To make sure there are no LLVM type errors when running the program, we use GADTs to define a type safe LLVM instruction set. We can generate this well types LLVM AST from the Accelerate code and then in the final step generate the actual LLVM code.

2.2.5 Irregular arrays

Because Accelerate is a higher level language than the CUDA code generated, there are quite some functionalities that can be implemented in CUDA but not easily in Accelerate. One of these things is the use of multi dimensional arrays, where not all sub-arrays have the same size. There has been research to add support for this in Accelerate using sequences of arrays [19]. These sequences are implemented by flattening the computations on them to regular computations. The transformation first has two steps where we transform the types. The first step normalizes arrays by adding an extra type that flattens nested arrays to a single larger dimensional array. The next step is vectorisation. This step transforms the type from the previous step in such a way that irregular arrays are changed to a single longer array with segments denoting where each sub-array starts. The actual program transformation works by removing all irregularity and changing it to regular computations. This uses a different rule for all different operations. For the actual executing of these computations the arrays are loaded into memory in chunks. The size of these chunks is determined dynamically.

2.3 Program transformation

To be able to add parallelism to a program, we need to transform the program. This is, however, not the only application where we need a program transformation. Almost all automatic optimizations to a program are defined in terms of a program transformation. Therefore, there has been research done to be able to efficiently define these transformations[20–23].

2.3.1 Ornaments

When implementing a program transformation we often want to be able to transform from one data type to the other. These data-types are often very similar. Usually, however, the relation between the two is not very easy to define. To make it easier to define we can use ornaments [24]. These ornaments work by relating a data-structure to a data-structure with the same recursive structure but with extra information added. For example we can relate natural number to lists by adding an extra element to the cons constructor. An ornament is defined as having a function from the more elaborate data-structure to the less elaborate one. This can easily be created by just forgetting the extra information the the more elaborate data-structure.

3 Methodology

We now know our goals, we however still need some more details about how to reach these goals. In this section we explain some of the basic concepts needed and we elaborate some more on our plans to reach our goals.

3.1 Parallelism

We need to be able to express the parallelism in the target language. For this, we use **fork** statements. These **fork** statements have two sub-statements that can be executed in parallel. In the implementation, when the program encounters a **fork** $\hat{t}_1$ $\hat{t}_2$ it creates a separate thread with $\hat{t}_1$ and continue executing $\hat{t}_2$. To handle the return values of these statements, we use references. These references refer to a mutable part of memory where multiple parts of the program can write to and read from. The reference itself is written to only once before forking the program and then the result is written to the part of memory the reference refers to. The rest of the program can then read the result from this part of memory when needed. We, however, need to be sure that the result has actually been computed up to the amount we need when we want to use the data from a reference. For this we use signals. These signals work in a way where if a piece of code is done updating some data in memory it resolves the signal. The code that then needs this data waits until the signal is resolved. This way we know for sure that the earlier code has finished before we use the data.

3.2 Parallelization strategy

To design our parallelization algorithm, we start by creating the inference rules for the transformation. We base these rules on the terms in the source language. In the rules we introduce forks to the program wherever possible. This means that we add forks when we encounter a term that has multiple sub-terms. In practice we add parallelism for let bindings and while loops.
To make these rules work in a more practical algorithm we also need some extra information, namely which variables are used in a sub-term. To make sure we do not analyze the terms multiple times for the same information we design an extra pass over the program to make sure all information is available wherever necessary. We can then formulate the actual transformation rules in terms of the statements in the source language so we can implement the transformation.

3.3 Optimizations

Our first basic parallelization results in significant overhead. It, for example, leads to a very large program, with a lot of forks. We can try to eliminate some of the forks. For example, there are forks that can never actually lead to extra parallelism because the second thread directly depends on the first thread. Removing these forks also means we can eliminate some of the signals. This makes the transformation doable for larger programs. Apart from eliminating forks, there are more optimizations that can be done. We analyze the results of the transformation to find some of these optimizations.

3.4 Formalization

As stated in Research Question 2 the goal is to have a good formalization of the transformation. To do this we define the transformation by creating an inference rule for each of the terms in the source language. This formalization has some simplifications. It should help to make the idea of the transformation easier to understand, and in extension also the resulting code.

3.5 Type safety

The Accelerate compiler is currently type safe, all variables have a specified type during the compilation. It is therefore good if our transformation is also able to keep this type safety, as stated in Research Question 3. During the transformation we need to keep track of the types of all variables. For this we use some of the strategies discussed in Section 2.2.4.

3.6 Benchmarking

For Research Question 4, we want to determine the speedup gained by the added parallelism. We want to look at both the compilation time and running time of the programs, as the compilation is also done at runtime. To determine the speedup we need to benchmark the implementation. To do this we use some Accelerate programs that we run with both the current implementation and our improved implementation. This shows us how much speedup we gained or lost with our addition of task parallelism.

3.7 Source language

To understand the format of the input language for the transformation, an AST representing the language from before the program transformation can be seen in Figure 1. An Accelerate program is always represented as a function f. This can directly be a term t or a lambda that takes in an argument. The term t has most of the standard control flow terms. Some notable things are the *op as*, which represents the performing of an operation with the given arguments. The **alloc** x represents allocating a buffer of size x. To use buffers we have **use** n *buf*, where we take n elements from buffer *buf*. The e represents simple expressions. In the implementation variables are represented using De Bruijn indices. In our representation however we simply use variable names.

$$
\begin{array}{rll}
vs & := () \mid x \mid (vs, vs) & \textit{Variables} \\
f & := t \mid \lambda x.f & \textit{Function} \\
t & := & \textit{Term} \\
 & \quad op\ vs & \textit{Operation} \\
 & \mid \textbf{return}\ vs & \textit{Return} \\
 & \mid e & \textit{Expression} \\
 & \mid \textbf{let}\ vs = t\ \textbf{in}\ t_1 & \textit{Let} \\
 & \mid \textbf{alloc}\ x & \textit{Allocation} \\
 & \mid \textbf{use}\ n\ buf & \textit{Use} \\
 & \mid \textbf{unit}\ v & \textit{Unit} \\
 & \mid \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2 & \textit{If} \\
 & \mid \textbf{while}\ f\ \textbf{do}\ f_1\ vs & \textit{While}
\end{array}
$$

Figure 1: A representation of the source language

3.8 Target language

The target language is similar to the source language. A representation can be seen in Figure 2. Some notable differences are that the terms are spread out over three different parts, one for the terms, one for the bindings and one for the effects. An effect *eff* represents a term that has a side effect, like waiting for or resolving signals, executing a kernel and writing to a reference. The bindings b represent statements that have a return value. These return values are not part of the general term, so these need to be handled separately. Additionally, we can see that the effect, **if** and **let** terms do not only have the parts needed for their corresponding functionality but also have the term that is executed after. Also the **while** itself is a bit different from the original one. Here it works by having a function $\hat{f}$ that takes in the current state of the loop and returns both whether or not the loop should continue and the result of the iteration. Finally the operation from before the transformation is compiled to the backend specific version of that operation. How this exact compilation works is not relevant for our research.

$vs :=$	$() \mid x \mid (vs, vs)$	*Variables*
$\hat{f} :=$	$t \mid \lambda x.f$	*Function*
$\hat{t} :=$		*Term*
	return	*Return*
	let $x = b$ **in** $\hat{t}$	*Let*
	eff ; $\hat{t}$	*Effect*
	if v **then** $\hat{t}_1$ **else** $\hat{t}_2$; $\hat{t}_3$	*If*
	while $\hat{f}$ vs ; $\hat{t}$	*While*
	fork $\hat{t}_1$ $\hat{t}_2$	*Fork*
$b :=$		*Binding*
	e	*Expression*
	signal	*Signal creation*
	ref	*Reference creation*
	alloc *sh vs*	*Allocation*
	use *n buf*	*Use*
	unit v	*Unit*
	read v	*Reference read*
eff :=		*Effect*
	kern vs	*Kernel*
	wait $[s]$	*Signal wait*
	resolve $[s]$	*Signal resolve*
	write v_1 v_2	*Reference write*

Figure 2: A representation of the target language

4 Basic transformation

To design the transformation we start by creating a basic version. The idea is that we add a fork whenever there are two sub-terms in a term, so we add forks for let bindings and while loops. Because we are using references to handle the possibly parallel executions, we also have to create new rules for the other statements in the language.

4.1 Basics

To define the transformation we create inference rules for every statement. In these inference rules we write the transformation as $\Gamma \vdash t \ @ \ d \rightsquigarrow \hat{t}$. Here, t is the term in the source language, which is transformed to $\hat{t}$ in the target language using environment Γ. Finally, d is the destination to write the result of the program to.

4.2 Environment

During the transformation we keep an environment with a mapping from variables in the source program to their corresponding reference and signals in the target program. In the implementation this environment also has the type of each of the variables. In the formalization, however, we simplify this by not adding the types. The environment should only contain variables that are used in the current program or explicitly removed from the environment.
The definition for the environment can be seen in Equation (4.1). Here, we represent the environment using a list where [] represents an empty environment and $\Gamma[v]$ represents variable v added to the environment Γ. For easier readability we also write $[v_1, \ldots, v_n]$ instead of $[][v_1] \ldots [v_n]$ for the environment containing exactly variables v_1 to v_n. Each of the variables v has a reference r, a lock l_r for reading and possibly a lock l_w for writing. More in this in the next subsection.

$$\Gamma = [] \mid \Gamma_1[x \mapsto r \ l_r] \mid \Gamma_1[x \mapsto r \ l_r \ l_w] \tag{4.1}$$

4.3 Destination passing style

The computation of a variable can be in a different thread from where it is used, the direct result might not be in scope where the variable is used. We, therefore, need to create a reference before forking so that this reference is in scope both for the computation and the usage. We, however, also need to be sure that the computation is actually finished before we can use the result. For this we use signals; when we are done computing a variable we resolve the signal. We can then wait for this signal before reading the reference. Multiple parts of the program can wait for the same signal but every signal has to be resolved exactly once. To represent the signals corresponding to a variable we use locks. Each lock $l = (s_w, s_r)$ contains two signals: one signal s_w to wait for before reading the value of the variable and one signal s_r to resolve when finished writing to the variable. In the implementation a lock does not need to have both these signals, as sometimes the signals are not needed. For simplicity, we assume that they are always both present.
We, however, not only change the variables in the program to use references. Also the result of the program uses references. Instead of returning a result, a destination reference should be passed to the program to which the result is written. The destination of the program is represented as a tuple of destination variables. One of these destinations is written as $(r \ s)$, where r is the reference to write the result and and s is the signal to resolve when finished.

4.4 Let

When we encounter a let binding in the program we introduce a fork. This fork parallelizes the term in the binding from the in-clause. This does mean that we need to introduce a reference for each of the variables we are binding here. We pass these references as the destination to the binding and in the environment to the **in** term. Since we are introducing a fork we also need to synchronize the variables that are used in both sub-terms. We denote the split of the environment as $\Gamma \twoheadrightarrow \Gamma_1 \bowtie \Gamma_2, s$, where Γ is split into Γ_1 and Γ_2 with s as the extra instruction for handling the signals. The details on how we do this can be found in Section 4.11.1. The resulting inference rule for a let-binding can be seen in Equation (4.2).

$$\frac{\begin{array}{c}\Gamma_1 \vdash t \mathbin{@} ((x'_1\ s_1), \ldots, (x'_n, s_n)) \rightsquigarrow \hat{t} \\ \Gamma_2[x_1 \mapsto x'_1\ (s_1, \_)\ (s_1, \_)] \ldots [x_n \mapsto x'_n\ (s_n, \_)\ (s_n, \_)] \vdash t_1 \mathbin{@} d \rightsquigarrow \hat{t}_1 \\ \Gamma \twoheadrightarrow \Gamma_1 \bowtie \Gamma_2, s\end{array}}{\Gamma \vdash \textbf{let}\ (x_1, \ldots x_n) = t\ \textbf{in}\ t_1 \mathbin{@} d \rightsquigarrow \begin{array}{l} s\ ; \\ \textbf{let}\ s_1 = \textbf{signal in} \\ \textbf{let}\ x'_1 = \textbf{ref in} \\ \vdots \\ \textbf{let}\ s_n = \textbf{signal in} \\ \textbf{let}\ x'_n = \textbf{ref in} \\ \textbf{fork}\ \hat{t}\ \hat{t}_1 \end{array}} \text{Let} \tag{4.2}$$

4.5 Binding

For the binding in a let we need to create a new let binding as we did not do that when encountering the let in the source program. The result of this let is then written to the destination. We, however, also possibly need to read the variables that are used from their corresponding reference. For this we first need to wait for the used variables to be ready. Then we can read these values. We should then also resolve the read signals as we ware done reading the variables. We can now perform the action of the binding. After this we should write the result to the destination and resolve the signals to tell that we are done writing. This results in the rules found in Equation (4.3).

$$\frac{}{[x_1 \mapsto r_1\ (s_1, s'_1),\ \ldots,\ x_n \mapsto r_n\ (s_n, s'_n)] \vdash e \mathbin{@} (r\ s) \rightsquigarrow \begin{array}{l} \textbf{wait}\ [s_1,\ \ldots,\ s_n] \\ \textbf{let}\ x_1 = \textbf{read}\ r_1\ \textbf{in} \\ \vdots \\ \textbf{let}\ x_n = \textbf{read}\ r_n\ \textbf{in} \\ \textbf{let}\ x = e\ \textbf{in} \\ \textbf{resolve}\ [s'_1,\ \ldots,\ s'_n] \\ \textbf{write}\ r\ x\ ; \\ \textbf{resolve}\ s\ ; \\ \textbf{return} \end{array}} \text{Expression} \tag{4.3a}$$

$$\frac{}{[x \mapsto r_1\ (s_1, s_2)] \vdash \textbf{alloc}\ x \mathbin{@} (r\ s) \rightsquigarrow \begin{array}{l} \textbf{wait}\ s_1\ ; \\ \textbf{let}\ x = \textbf{read}\ r_1\ \textbf{in} \\ \textbf{let}\ xs = \textbf{alloc}\ x\ \textbf{in} \\ \textbf{resolve}\ s_2\ ; \\ \textbf{write}\ r\ xs\ ; \\ \textbf{resolve}\ s\ ; \\ \textbf{return} \end{array}} \text{Alloc} \tag{4.3b}$$

$$\frac{}{[] \vdash \textbf{use}\ n\ buf\ @\ (r\ s) \rightsquigarrow \begin{array}{l}\textbf{let}\ x = \textbf{use}\ n\ buf\ \textbf{in}\\ \textbf{write}\ r\ x\ ;\\ \textbf{resolve}\ s\ ;\\ \textbf{return}\end{array}}\ \text{Use} \tag{4.3c}$$

$$\frac{}{[x \mapsto r_1\ (s_1, s_2)] \vdash \textbf{unit}\ x\ @\ (r\ s) \rightsquigarrow \begin{array}{l}\textbf{wait}\ s_1\ ;\\ \textbf{let}\ x = \textbf{read}\ r_1\ \textbf{in}\\ \textbf{let}\ xs = \textbf{unit}\ x\ \textbf{in}\\ \textbf{resolve}\ s_2\ ;\\ \textbf{write}\ r\ xs\ ;\\ \textbf{resolve}\ s\ ;\\ \textbf{return}\end{array}}\ \text{Unit} \tag{4.3d}$$

4.6 Function

Functions are a special case as they are not a regular term in the program and do not have a destination. We therefore use a slightly altered notation for the transformation without the destination. A lambda is handled by adding the argument to the environment. For a body we need to create a new function with an argument for the references of the output to be passed to. We can then use these references as the destination of the content of the body. The rules for this can be seen in Equation (4.4).

$$\frac{\Gamma[x] \vdash f \rightsquigarrow \hat{f}}{\Gamma \vdash \lambda x.f \rightsquigarrow \lambda x.\hat{f}}\ \text{Lambda} \tag{4.4a}$$

$$\frac{\Gamma[x] \vdash t\ @\ (r\ s) \rightsquigarrow \hat{t}}{\Gamma \vdash t \rightsquigarrow \lambda(r, s).\hat{t}}\ \text{Body} \tag{4.4b}$$

4.7 Condition

For a condition we need the program to wait for the guard variable to be ready, so we can read the value from the reference. We can recursively convert both branches. Since we have two branches that do not necessarily use the same variables, we have to do some synchronization for the environments. For example if a variable is used in only one of the branches we need to resolve the signals for this variable in the other branch. For this we use the separate sub-environment rule in Equation (4.6) that removes the variables from the environment that are not used and resolves their signals. How the synchronization is defined exactly can be seen in Section 4.11.2. This results in the rule for conditions in Equation (4.5).

$$\frac{\Gamma \vdash t_1\ @\ d \rightsquigarrow \hat{t}_1 \quad \Gamma \vdash t_2\ @\ d \rightsquigarrow \hat{t}_2 \quad \Gamma(v) = r\ (s,\ \_)}{\Gamma \vdash \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ @\ d \rightsquigarrow \begin{array}{l}\textbf{wait}\ s\ ;\\ \textbf{if}\ v\\ \textbf{then}\ \hat{t}_1\\ \textbf{else}\ \hat{t}_2\ ;\\ \textbf{return}\end{array}}\ \text{If} \tag{4.5}$$

$$\frac{\Gamma' \vdash t\ @\ d \rightsquigarrow \hat{t} \quad \Gamma \supseteq \Gamma', s}{\Gamma \vdash t\ @\ d \rightsquigarrow \textbf{resolve}\ s\ ;\ \hat{t}}\ \text{Sub-environment} \tag{4.6}$$

4.8 Operation

An operation needs to be compiled to a kernel that can be executed on the target device. This is done in this step, but how it is done exactly is not part of this thesis. The operation does use variables, both for reading input and for writing the output. We, thus, need to wait for these used variables to be ready and read them from their references. After this we can resolve the signals for the used variables.

$$\frac{\Gamma = [x_1 \mapsto r_1\ (s_1^1, s_1^2)\ (s_1^3, s_1^4), \ldots, x_n \mapsto r_n\ (s_n^1, s_n^2)\ (s_n^3, s_n^4)] \quad op \Rightarrow kern}{\Gamma \vdash op\ (x_1, \ldots, x_n)\ @\ () \rightsquigarrow \begin{array}{l} \textbf{wait}\ [s_1^1, s_1^3, \ldots, s_n^1, s_n^3]\ ; \\ \textbf{let}\ x_1 = \textbf{read}\ r_1\ \textbf{in} \\ \vdots \\ \textbf{let}\ x_n = \textbf{read}\ r_n\ \textbf{in} \\ \textbf{Exec}\ kern\ (x_1, \ldots, x_n)\ ; \\ \textbf{resolve}\ [s_1^2, s_1^4, \ldots, s_n^2, s_n^4]\ ;\ \textbf{return} \end{array}} \text{ Operation} \tag{4.7}$$

4.9 Return

As we now use references for the output of a program we need to write the output when we encounter a return statement. We do this by first waiting for the result to be ready. After this we read this result from their corresponding reference, after which we can write this result to the output reference. Finally we can resolve the output signals.

$$\frac{\Gamma = [x_1 \mapsto r_1\ (s_1, s_1'),\ \ldots,\ x_n \mapsto r_n\ (s_n, s_n')]}{\Gamma \vdash \textbf{return}\ (x_1, \ldots, x_n)\ @\ ((r_1'\ s_1''), \ldots, (r_n'\ s_n'')) \rightsquigarrow \begin{array}{l} \textbf{wait}\ [s_1, \ldots, s_n]\ ; \\ \textbf{let}\ x_1 = \textbf{read}\ r_1\ \textbf{in} \\ \textbf{write}\ r_1'\ x_1\ ; \\ \vdots \\ \textbf{let}\ x_n = \textbf{read}\ r_n\ \textbf{in} \\ \textbf{write}\ r_n'\ x_n\ ; \\ \textbf{resolve}\ [s_1', s_1'', \ldots, s_n', s_n'']\ ; \\ \textbf{return} \end{array}} \text{ Return} \tag{4.8}$$

4.10 While

The **while** statement is probably the most complicated statement in the language. It is also another place where we add task parallelism. Here it is, however, not done by adding a **fork** statement. In this case it is the scheduler that can decide to start on the next iteration before the previous one has started. In practice this means that after the guard has been computed and we know that there is another iteration, there might already be another thread starting on the next iteration while the current iteration is still being preformed. In the transformation we have to combine the two functions from before the transformation, for the guard and the body, into a single function. This new function takes in the result of the previous iteration and returns the result of the guard and the result of the iteration. As we are using destination passing style these outputs are references pasted to the function. In the function we first create a destination for the guard function and then execute this guard. After this we can write the result of the guard to it's output reference. If the guard is true we perform an iteration of the while loop with as destination the reference for the result of the iteration. If the guard was false we write the current values of the while loop to the original destination.

$$\frac{\Gamma[x \mapsto r\ (s, \_)] \vdash t\ @\ (r_2\ s_2) \rightsquigarrow \hat{t} \quad \Gamma[x \mapsto r\ (s, \_)] \vdash t_1\ @\ (r_1\ s_1) \rightsquigarrow \hat{t}_1 \quad [y \mapsto r_1\ (s_1, \_)] \vdash \textbf{return}\ y\ @\ d \rightsquigarrow \hat{t}'}{\Gamma \vdash \textbf{while}\ \lambda x.t\ \textbf{do}\ \lambda x.t_1\ vs\ @\ d \rightsquigarrow \begin{array}{l} \textbf{while} \\ (\lambda(r, s).\lambda(r_0, s_0).\lambda(r_1, s_1). \\ \textbf{let}\ s_2 = \textbf{signal in} \\ \textbf{let}\ r_2 = \textbf{ref in} \\ \hat{t} \\ \textbf{wait}\ [s_2] \\ \textbf{let}\ g = \textbf{read}\ r_2\ \textbf{in} \\ \textbf{write}\ r_0\ g \\ \textbf{resolve}\ [s_0] \\ \textbf{if}\ g\ \textbf{then}\ \hat{t}_1 \\ \textbf{else}\ \hat{t}') \\ vs\ ;\ \textbf{return} \end{array}} \text{While} \tag{4.9}$$

4.11 Environment synchronization

In some of the rules we need to synchronize the variables in the environment. In this subsection we discuss how we do this for the different occasions.

4.11.1 Parallel

When introducing a fork we have to split the environment for the two different threads. Since these threads can be executed in parallel we also have to do some extra synchronization to make sure all the used values are actually computed before using them. In general a read has to wait for all previous writes to be finished and a write has to wait until all previous writes and reads are finished. To do this we regularly have to create new signals which can then be resolved in one thread and waited on in the other. Some of the signals that need to be resolved depend on both of the threads. In this case we create an extra thread that waits for the signals in the two threads and then resolves the signal that need to be resolved. The resulting rules can be seen in Equation (4.10).
In Equation (4.10c), when both threads only read the variable, we let both threads wait for the original signal and create a new separate thread that resolves the signal for the variable when both threads are done reading. When the first thread reads and the second threads writes in Equation (4.10d) we need the second to wait with reading until the first thread is done writing to the variable. We also need to wait with resolving the original write signal, so we create a separate thread to resolve it. In Equation (4.10e), when the first thread reads and the second thread writes, we have to create a thread for the second thread to wait with writing until the variable is ready to be written to from before this point and the first thread is done reading. Finally when both threads write to the variable in Equation (4.10f) the second thread has to wait for the first thread to be finished reading before it can write and finished writing before it can read.

$$\frac{}{[] \twoheadrightarrow [] \bowtie [], id}\ \text{Empty} \tag{4.10a}$$

$$\frac{\Gamma_1 \rightsquigarrow \Gamma_1 \bowtie \Gamma_2,\ \hat{t} \quad x \twoheadrightarrow x_1 \bowtie x_2,\ \hat{t}_1}{\Gamma[x] \rightsquigarrow \Gamma_1[x_1] \bowtie \Gamma_2[x_2],\ \hat{t}_1\ .\ \hat{t}}\ \text{Cons} \tag{4.10b}$$

$$\frac{}{x \mapsto r\ (s_1^r, s_2^r) \twoheadrightarrow x \mapsto r\ (s_1^r, s_1) \bowtie x \mapsto r\ (s_1^r, s_2),\ \begin{array}{l}\textbf{newSignal}\ s_1\\ \textbf{newSignal}\ s_2\\ \textbf{waitResolve}\ [s_1, s_2]\ s_2^r\end{array}}\ \text{Read-read} \tag{4.10c}$$

$$\frac{}{x \mapsto r\ (s_1^r, s_2^r)\ (s_1^w, s_2^w) \twoheadrightarrow x \mapsto r\ (s_1^r, s_1)\ (s_1^w, s_2) \bowtie x \mapsto r\ (s_2, s_3),\ \begin{array}{l}\textbf{newSignal}\ s_1\\ \textbf{newSignal}\ s_2\\ \textbf{newSignal}\ s_3\\ \textbf{waitResolve}\ [s_1, s_3]\ s_2^r\\ \textbf{waitResolve}\ [s_2]\ s_2^w\end{array}}\ \text{Write-read} \tag{4.10d}$$

$$\frac{}{x \mapsto r\ (s_1^r, s_2^r)\ (s_1^w, s_2^w) \twoheadrightarrow x \mapsto r\ (s_1^r, s_1) \bowtie x \mapsto r\ (s_1^r, s_2^r)\ (s_2, s_2^w),\ \begin{array}{l}\textbf{newSignal}\ s_1\\ \textbf{newSignal}\ s_2\\ \textbf{waitResolve}\ [s_1^w, s_1]\ s_2\end{array}}\ \text{Read-write} \tag{4.10e}$$

$$\frac{}{x \mapsto r\ (s_1^r, s_2^r)\ (s_1^w, s_2^w) \twoheadrightarrow x \mapsto r\ (s_1^r, s_1)\ (s_1^w, s_2) \bowtie x \mapsto r\ (s_2, s_2^r)\ (s_1, s_2^w),\ \begin{array}{l}\textbf{newSignal}\ s_1\\ \textbf{newSignal}\ s_2\end{array}}\ \text{Write-write} \tag{4.10f}$$

$$\textbf{newSignal}\ s := \textbf{let}\ s = \textbf{signal}\ \textbf{in} \qquad \text{(New signal)}$$

$$\textbf{waitResolve}\ s_1\ s_2 := \textbf{fork}\ (\textbf{wait}\ s_1\ ;\ \textbf{resolve}\ s_2\ ;\ \textbf{return}) \qquad \text{(Wait resolve)}$$

4.11.2 Conditional

In a condition we have two sub-terms of which only one is executed. These sub-terms do not always use the same variables. We, however, always need to resolve the signals of a variable. We need to be able to reduce the environment to only the variables that are needed and resolve the not used signals, so if a variable is in the environment but not used we remove it from the environment and resolve all the signals for that variable. However if the variable is in the environment with write privileges but we only read the variable we also need to resolve the write signal of the variable. The rules for this can be seen in Equation (4.11).

$$\frac{}{[] \supseteq [], []} \tag{4.11a}$$

$$\frac{x \geq x', s}{\Gamma[x] \supseteq \Gamma[x'], s} \tag{4.11b}$$

$$\frac{}{x \mapsto r\ (s_1, s_2)(s_3, s_4) \geq x \mapsto r\ (s_1, s_2)(s_3, s_4), []} \tag{4.11c}$$

$$\frac{}{x \mapsto r\ (s_1, s_2)(s_3, s_4) \geq x \mapsto r\ (s_1, s_2), [s_4]} \tag{4.11d}$$

$$\frac{}{x \mapsto r\ (s_1, s_2)(s_3, s_4) \geq (), [s_2, s_4]} \tag{4.11e}$$

$$\frac{}{x \mapsto r\ (s_1, s_2) \geq x \mapsto r\ (s_1, s_2), []} \tag{4.11f}$$

$$\frac{}{x \mapsto r\ (s_1, s_2) \geq (), [s_2]} \tag{4.11g}$$

(4.11h)

4.12 Usage analysis

In order to decide what rule to use for the synchronizations in Section 4.11 we need to know what variables are used in each sub-term. To do this we have an extra pass over the program before the transformation where we annotate this. The rules for the analysis can be seen in Equation (4.12). This analysis gives the variables that are used in a term and whether these variables are only read from or also written to. We then annotate these result to each term in the program, so we can use this for the synchronization.

$$\frac{f : us}{\lambda x.f : us \backslash x}\ \text{Lambda} \tag{4.12a}$$

$$\frac{}{op\ vs : vs}\ \text{Operation} \tag{4.12b}$$

$$\frac{}{\textbf{return}\ vs : [vs \rightarrowtail R]}\ \text{Return} \tag{4.12c}$$

$$\frac{t : us_1 \quad t_1 : us_2}{\textbf{let}\ x = t\ \textbf{in}\ t_1 : us_1 \cup us_2 \backslash x}\ \text{Let} \tag{4.12d}$$

$$\frac{t_1 : us_1 \quad t_2 : us_2}{\textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : [v \rightarrowtail R] \cup us_1 \cup us_2}\ \text{If} \tag{4.12e}$$

$$\frac{f : us_1 \quad f_1 : us_2}{\textbf{while}\ f\ \textbf{do}\ f_1\ vs : us_1 \cup us_2}\ \text{While} \tag{4.12f}$$

$$\frac{}{e : []}\ \text{Expression} \tag{4.12g}$$

$$\frac{}{\textbf{alloc}\ vs : [vs \rightarrowtail R]}\ \text{Alloc} \tag{4.12h}$$

$$\frac{}{\textbf{use}\ n\ buf : []}\ \text{Use} \tag{4.12i}$$

$$\frac{}{\textbf{unit}\ v : [v \rightarrowtail R]}\ \text{Unit} \tag{4.12j}$$

(4.12k)

4.13 Examples

Now that we have formulated all the rules for the basic transformation, we can look at some examples. In Example 4.13.1 we have a simple example of only a single operation. Example 4.13.2 contains a little bit longer example that has two operations that do not depend on each other, so here a fork would be able to improve the performance. In both these examples we can clearly see that we are introducing a lot of extra terms to the program with extra signals, references and forks. In the next section we look into eliminating some of these forks as not all of them are actually able to lead to a program speedup.

Example 4.13.1. We look at a simple input program that takes in an array and adds one to every element of the array.

```
λ (e0, b0).
    let b1 = alloc e0 in
    let () = map (\x0 -> 1.0 + x0, in b0, out b1) in
    return (e0, b1)
```

This result in the following program after transformation:

```
λ ((s0, r0), (s1, r1)).                  -- Signals and references for the input
    λ ((s2, r2), (s3, r3)).              -- Signals and references for the output
        let s4 = signal in               -- Read and write signal for the result
        let s5 = signal in
        let r4 = ref in                  -- Reference to write the alloc result to
        fork {                           -- Forking the alloc from the rest
            let s6 = signal in           -- Read signal for map result
            fork {                       -- Fork map from the return
                wait [s0, s6]            -- Wait for result to be ready
                let e2 = read r0 in      -- Read the input size
                write r2 e2              -- Write size to the output ref
                let b3 = read r4 in      -- Read map result
                write r3 b3              -- Write result to output
                resolve [s2, s3]         -- Resolve output signals
            }
            wait [s0, s1, s4, s5]        -- Wait for input array and allocation
            let e1 = read r0 in          -- Read input array size
            let b1 = read r1 in          -- Read input array
            let b2 = read r4 in          -- Read buffer to write result to
            map (\x0 -> 1.0 + x0, in b1, out b2) -- Perform the map
            resolve [s6]                 -- Resolve the read signal
        }
        wait [s0]                        -- Wait for input size to be available
        let e0 = read r0 in              -- Read the input size reference
        let b0 = alloc e0 in             -- Perform the allocation
        write r4 b0                      -- Write result to output
        resolve [s4, s5]                 -- Resolve read and write signals
```

△

Example 4.13.2. We can now also look at a bit more complicated example. This example takes in an array and outputs a tuple with two arrays. The first output array is the input array but every element has one added to it. The second output array is an array of length two with the second and third element of the input array.

```
λ (e0, b0).
    let b1 = alloc e0 in
    let () = map (\x0 -> 1.0 + x0, in b0, out b1) in
    let e1 = 2 in
    let b2 = alloc e1 in
    let () = backpermute (\x0 -> 1 + x0, in b0, out b2) in
    return ((e0, b1), (e1, b2))
```

After transformation we get the following program:

```
λ ((s0, r0), (s1, r1)).
    λ (((s2, r2), (s3, r3)), ((s4, r4), (s5, r5))).
        let s6 = signal in
        let s7 = signal in
        let r6 = ref in
        fork {
            let s8 = signal in
            fork {
                let s9 = signal in
                let r7 = ref in
                fork {
                    let s10 = signal in
                    let s11 = signal in
                    let r8 = ref in
                    fork {
                        let s12 = signal in
                        fork {
                            wait [s0, s8, s9, s12]
                            let e4 = read r0 in
                            write r2 e4
                            let b6 = read r6 in
                            write r3 b6
                            let e5 = read r7 in
                            write r4 e5
                            let b7 = read r8 in
                            write r5 b7
                            resolve [s2, s3, s4, s5]
                        }
                        wait [s0, s1, s10, s11]
                        let b4 = read r1 in
                        let b5 = read r8 in
                        backpermute (\x0 -> 1 + x0, in b4, out b5)
                        resolve [s12]
                    }
                    wait [s9]
                    let e3 = read r7 in
                    let b3 = alloc e3 in
                    write r8 b3
                    resolve [s10, s11]
```

```
        }
        let e2 = 2 in
        write r7 e2
        resolve [s9]
    }
    wait [s0, s1, s6, s7]
    let e1 = read r0 in
    let b1 = read r1 in
    let b2 = read r6 in
    map (\x0 -> 1.0 + x0, in b1, out b2)
    resolve [s8]
}
wait [s0]
let e0 = read r0 in
let b0 = alloc e0 in
write r6 b0
resolve [s6, s7]
```

△

5 Optimizations

Now that we have formulated the basic rules we can look at optimizing the transformation. In this section we look at removing forks that do not give any extra performance and we add some extra forks for returning the variables.

5.1 Trivial

Some of the terms we are currently parallelizing are so small that parallelizing them does not yield any speedup. For example the compute 2 from Example 4.13.2. So to improve the transformation we are not parallelizing these trivial terms. The terms that we call trivial are **use** and **compute** with an expression that does not use any variable. We could say that an **alloc** and **unit** would also be trivial since the execution is also basically instant, but because they use variables the waiting for the variable to be ready might take a non-trivial amount of time.
Since these terms are not parallelized, we know that the result is ready when it is used. This also means that we do not need a reference or any signals. To implement this we have a separate rule for a let binding where we keep it as a let binding in the transformation, as seen in Equation (5.1). This rule is used when the term that is bound in the let is one of the trivial terms. In this rule we transform the binding t to the corresponding binding b in the target language. As we do not use references this transformation does need need a destination. The transformation is very close to an identity transformation as there are no extra things that we need to do.
Not using a reference for these variables does also mean that we have to change some of the other rules as well. Because now we do not need to read the value from the reference before using the value. This is a quite trivial change, so we do not elaborate on these rules here.

$$\frac{\Gamma_1 \vdash t \rightsquigarrow b \quad \Gamma_2 \vdash t_1 \mathbin{@} d \rightsquigarrow \hat{t}_1}{\Gamma \vdash \textbf{let } x = t \textbf{ in } t_1 \mathbin{@} d \rightsquigarrow \textbf{let } x = b \textbf{ in } \hat{t}_1} \text{ Let-sequential} \tag{5.1}$$

5.2 Directly waiting

Usually, the result of a statement is used in another statement. If it is the next statement that uses it, there is no use in forking these two statements from each other, since the second statement waits for the first one to finish in either case. We can, however, also not simply eliminate the fork for the first statement as there might be more statement later that do not depend on either statement, which could be executed in parallel. We want to combine the two statements into a single thread, which is then forked from the rest of the program. To implement this we look at the first statement in the in-clause of a let binding and see if it uses all results from the let binding. If it does we use the rule in Equation (5.2) to combine the current binding with the first statement in the in-clause. Here the $t'' \leftarrow t'\ t'_1$ puts the t' directly in the first fork it encounters in t'_1.

$$\frac{\Gamma_1 \vdash t \mathbin{@} (x'\ s_1) \rightsquigarrow \hat{t} \quad \Gamma_2[x \mapsto x'(s_1, \_)(s_1, \_)] \vdash t_1 \mathbin{@} d \rightsquigarrow \hat{t}_1 \quad \Gamma \rightarrowtail \Gamma_1 \bowtie \Gamma_2, s \quad \hat{t}' \leftarrow \hat{t}\ \hat{t}_1}{\Gamma \vdash \textbf{let } x = t \textbf{ in } t_1 \mathbin{@} d \rightsquigarrow s\ ; \textbf{let } s_1 = \textbf{signal in let } x' = \textbf{ref in } \hat{t}'} \text{ Let-combine} \tag{5.2}$$

5.3 Fork returns

Often, a return statement has multiple values to return, like in Example 4.13.2. In the basic transformation we simply write these results sequentially. It might, however, be that the different values to return are ready at different times. It can, therefore, be better to write these results in parallel, so that if one is ready earlier it is already returned and can be used somewhere else. To do this we introduce new forks for every value that is to be returned. The rules for this can be seen in Equation (5.3).

$$\frac{}{[] \vdash \textbf{return}\ ()\ @\ () \rightsquigarrow \textbf{return}}\ \text{Return} \tag{5.3a}$$

$$\frac{\Gamma_1 \vdash \textbf{return}\ vs_1\ @\ d_1 \rightsquigarrow \hat{t}_1 \quad \Gamma_2 \vdash \textbf{return}\ vs_2\ @\ d_2 \rightsquigarrow \hat{t}_2 \quad \Gamma = \Gamma_1 \cup \Gamma_2}{\Gamma \vdash \textbf{return}\ (vs_1, vs_2)\ @\ (d_1, d_2) \rightsquigarrow \textbf{fork}\ \hat{t}_1\ \hat{t}_2}\ \text{Return} \tag{5.3b}$$

$$\frac{}{[v \mapsto r_1\ (s_1, s_2)] \vdash \textbf{return}\ v\ @\ (r\ s) \rightsquigarrow \begin{array}{l}\textbf{wait}\ s_1\ ; \\ \textbf{let}\ x = \textbf{read}\ r_1\ \textbf{in} \\ \textbf{write}\ r\ x\ ; \\ \textbf{resolve}\ s_2\ \texttt{++}\ s\ ;\ \textbf{return}\end{array}}\ \text{Return} \tag{5.3c}$$

5.4 Example

With these optimizations added to the transformation the example from Example 4.13.2 results in the program found in Example 5.4.1.

Example 5.4.1.

```
λ ((s0, r0), (s1, r1)).
    λ (((s2, r2), (s3, r3)), ((s4, r4), (s5, r5))).
        let s6 = signal in
        let s7 = signal in
        let r6 = ref in
        let s8 = signal in
        fork {
            let e2 = 2 in
            let s10 = signal in
            let s11 = signal in
            let r8 = ref in
            let s12 = signal in
            fork {
                fork {
                    fork {
                        wait [s0]
                        let e4 = read r0 in
                        write r2 e4
                        resolve [s2]
                    }
                    wait [s8]
                    let b6 = read r6 in
                    write r3 b6
                    resolve [s3]
                }
                fork {
                    write r4 e2
                    resolve [s4]
                }
```

```
        wait [s12]
        let b7 = read r8 in
        write r5 b7
        resolve [s5]
    }
    let b3 = alloc e2 in
    write r8 b3
    resolve [s10, s11]
    wait [s0, s1, s10, s11]
    let b4 = read r1 in
    let b5 = read r8 in
    backpermute (\x0 -> 1 + x0, in b4, out b5)
    resolve [s12]
}
wait [s0]
let e0 = read r0 in
let b0 = alloc e0 in
write r6 b0
resolve [s6, s7]
wait [s0, s1, s6, s7]
let e1 = read r0 in
let b1 = read r1 in
let b2 = read r6 in
map (\x0 -> 1.0 + x0, in b1, out b2)
resolve [s8]
```

△

6 Results

Now that we have defined the transformation we can look at how well the transformation works in practice. In this section we look at the implementation and the tests we ran on this.

6.1 Implementation

To see how well the transformation works in practice we created an implementation based on the transformation rules[1]. We try to stay as close as possible to the formal definition to make the implementation easier to understand and reason about. The implementation replaces an existing transformation for task parallelism in the compile pipeline. In the pipeline all terms in the AST's have the types annotated. We want to make sure that these types are also preserved during our transformation. So in the implementation we keep track of the typed environment with the current variables and their corresponding types.

6.2 Benchmarks

We also want to test the speed of our implementation. To do this we run some benchmarks. These benchmarks are ran using the interpreter backend on an AMD Ryzen 9 7950X with 64GB of DDR5-5600 memory. Every value is an average over ten runs. We test three different implementations. The current version of Accelerate with only a very basic implementation of task parallelism, the existing implementation of the new pipeline that is being developed with an implementation for task parallelism similar to ours and finally our own implementation for task parallelism, which is based on the new pipeline where we replace the existing transformation for task parallelism with ours.
We use two different programs for testing. The first one is an implementation of an automatic differentiation benchmark[25] in Accelerate. This program is quite complicated with many different kernels. The second one is a salt marsh creek formation benchmark. This benchmark is a simulation of creek formation in a salt marsh ecosystem, using mainly stencil operations.

6.2.1 Compilation time

We first run some benchmarks to test the compilation time of the different implementations. The results for the automatic differentiation benchmark can be found in Figure 3. We can see that the new pipeline is a lot slower than the current Accelerate pipeline. A major part of this is an analysis for strongly live variables. This analysis does a lot of weakening of variable indices. The existing implementation for task parallelism that we replaced with ours also does this analysis another time, this is, thus, also why our implementation is significantly faster.
The result for the salt marsh benchmarks can be found in Figure 4. We can see that here the new pipeline is not a lot slower. This is most likely because the program is a lot easier, so the weakening is usually done on only a handful of variables. We can, however, still see that the new pipeline is a bit slower that the current version of Accelerate. We also see that our implementation is somewhere in between the two.
What we can conclude from these results is that our implementation most likely does not add a lot of extra time to the compilation, which is what we aimed for.

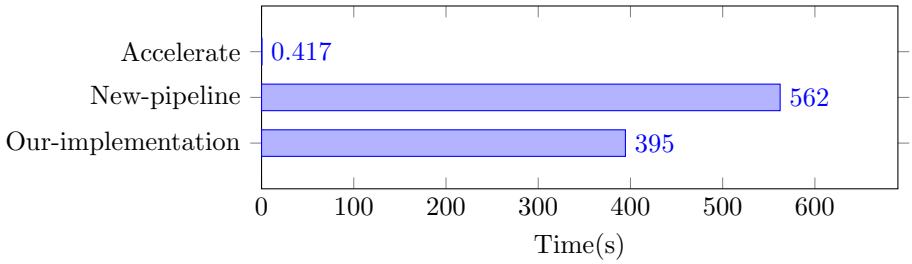


Figure 3: Compilation time of the automatic differentiation benchmark program.

[1] https://github.com/musicismyalibi/accelerate/blob/task-parallelism/src/Data/Array/Accelerate/Trafo/Schedule/Uniform.hs

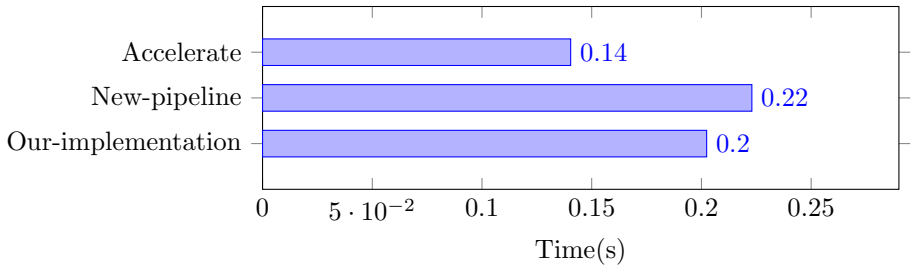


Figure 4: Compilation time of the salt marsh benchmark program.

6.2.2 Running time

We also want to test the speedup of the actual program. For this we use the salt marsh benchmark. We use the interpreter backend as the other backends do not work correctly with the new pipeline version of Accelerate. The interpreter is designed to simply be able to run an Accelerate program, but without the real parallel computations. We are, therefore, not really able to test what kind of speedup our addition of task parallelism gives, but only an indication of the amount of overhead we add. The backend for the new pipeline does support some parallelization.
We first test the runtime where we force the program to use a single thread. The results can be found in Figure 5. We can see that the new pipeline with the extra support of forks and references does make the runtime quite a bit longer. We can also see that our implementation adds a little bit less overhead than the implementation in the new pipeline.
We can now have a quick look at the running time for multiple threads. The result can be found in Figure 6. We can see that these result are quite the opposite of what we would expect, as the program becomes slower when using more threads. We can however see that this pattern exists for all our compiler versions. This is most likely caused by some issues with running Haskell in parallel, where it forces running on multiple threads even though the program itself does not really support this, causing a lot of extra unnecessary overhead.

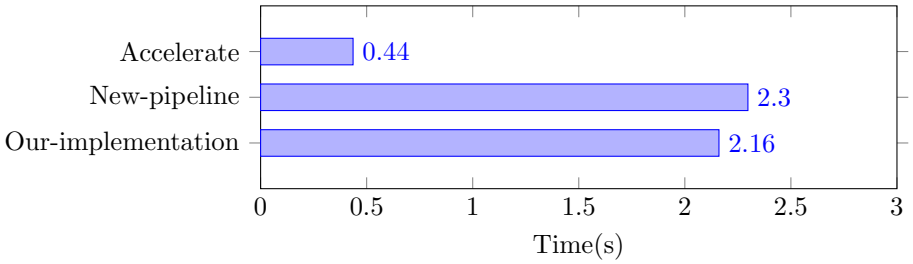


Figure 5: Running time of the salt marsh benchmark program on a single thread.

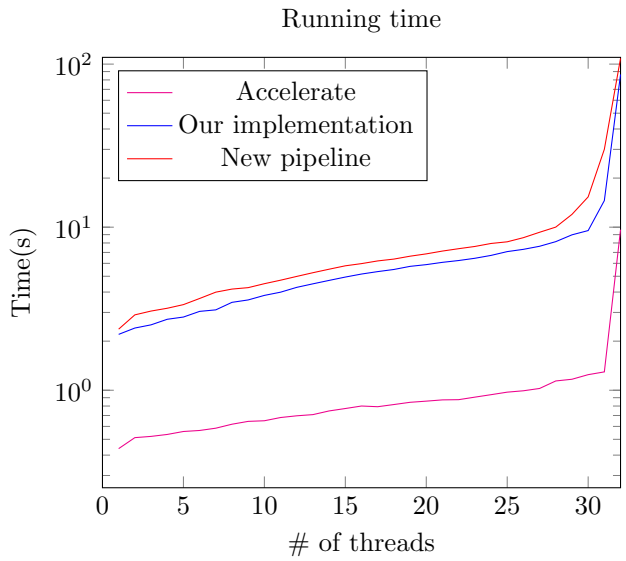


Figure 6: Running time of the salt marsh benchmark program on a multiple threads.

7 Conclusion

We have seen that by forking the program when encountering a let binding we can add task parallelism to a program. We have defined the transformation to do this by creating inference rules for every statement in the language. To improve the transformation we optimized it by not forking trivial statements, combining statements when they wait on the previous one and adding extra forks when returning multiple variables.
We have then implemented the transformation based on the definition. This implementation has been made to be type safe. We found that the implementation in both compilation and single threaded runtime is slower than the current Accelerate implementation, but faster that the existing implementation for task parallelism in the new pipeline.

7.1 Research questions

We can now look at the answers to our research questions from Section 1.3. For question 2 we have seen that we were able to create a good formalization of the transformation by creating inference rules for each term in the source language. For question 3 we have seen that in the implementation we were able to keep the same type safety as in the rest of the Accelerate compiler by keeping track of the types in the environment. For question 4 we do not really know the answer yet as we did not have the time to properly test the runtime of the programs in a parallel manner.
So in conclusion for question 1 we have seen that we can add task parallelism to Accelerate in quite a well defined, type safe and efficient way by first creating inference rules for the transformation and then creating an implementation based on this.

8 Discussion

We have created a good transformation to add task parallelism to accelerate. There are, however, still ways to improve on our research in future work. In this section we discuss some of these.

8.1 Benchmarks

The benchmarks we ran for this research were quite limited. We only tested two different programs and only for the interpreter backend. It would be good to do some more tests to see how it behaves for more different programs. It would also bee good to test whether the addition of task parallelism will actually make the program execution faster when ran in parallel.

8.2 Optimizations

We have already added some optimizations to the transformation. There are, however, many more optimizations possible that could improve the performance of the program. These optimizations will most likely mostly be about removing overhead in the form of references and signals. An easy improvement could be to remove some of the signals when combining two statements into a single thread. More broadly it would most likely be possible to combine more statements together and remove extra references and signals when they are not needed. Apart from this it might be possible to treat some more statements as trivial when we already know that the variables they use will for certain be ready.

8.3 Graph based

In this research we decided to introduce forks mostly in let bindings. This does however mean that we will, in the basis, only parallelize single terms in a let binding. It could however be beneficial to parallelize series of terms that directly depend on each other. To find these we could create some sort of graph with all dependencies of the program. This graph can then be used to decide where to introduce forks. These forks can then more easily have two larger subterms to parallelize. It could be good to look at this different method of automatic task parallelism in the future.

References

[1] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011, pp. 3–14.

[2] Barry Bond, Kerry Hammil, Lubomir Litchev, and Satnam Singh. "FPGA Circuit Synthesis of Accelerator Data-Parallel Programs". In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 2010, pp. 167–170. DOI: 10.1109/FCCM.2010.51.

[3] Thomas Jansen. "GPU++ - An Embedded GPU Development System for General-Purpose Computations". en. PhD thesis. Technische Universität München, 2007, p. 144.

[4] Geoffrey Mainland and Greg Morrisett. "Nikola: Embedding Compiled GPU Functions in Haskell". In: *SIGPLAN Not.* 45.11 (Sept. 2010), pp. 67–78. ISSN: 0362-1340. DOI: 10.1145/2088456.1863533. URL: https://doi.org/10.1145/2088456.1863533.

[5] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, AD Sarma, D Nanongkai, G Pandurangan, P Tetali, et al. "PyCUDA: GPU run-time code generation for high-performance computing". In: *Arxiv preprint arXiv* 911 (2009).

[6] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. "Shader Algebra". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: Association for Computing Machinery, 2004, pp. 787–795. ISBN: 9781450378239. DOI: 10.1145/1186562.1015801. URL: https://doi.org/10.1145/1186562.1015801.

[7] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. "Automatic program parallelization". In: *Proceedings of the IEEE* 81.2 (1993), pp. 211–243. DOI: 10.1109/5.214548.

[8] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. "Automatic generation of nested, fork-join parallelism". In: *The Journal of Supercomputing* 3.2 (1989), pp. 71–88.

[9] Guido Hogen, Andrea Kindler, and Rita Loogen. "Automatic parallelization of lazy functional programs". In: *European Symposium on Programming*. Springer. 1992, pp. 254–268.

[10] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. "Effective Automatic Parallelization of Stencil Computations". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 235–244. ISSN: 0362-1340. DOI: 10.1145/1273442.1250761. URL: https://doi.org/10.1145/1273442.1250761.

[11] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. "Software Behavior Oriented Parallelization". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 223–234. ISSN: 0362-1340. DOI: 10.1145/1273442.1250760. URL: https://doi.org/10.1145/1273442.1250760.

[12] Alexander Aiken and Alexandru Nicolau. "Optimal loop parallelization". In: *ACM SIGPLAN Notices* 23.7 (1988), pp. 308–317.

[13] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. "Polaris: Improving the effectiveness of parallelizing compilers". In: *Languages and Compilers for Parallel Computing*. Ed. by Keshav Pingali, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 141–154. ISBN: 978-3-540-49134-7.

[14] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://doi.org/10.1145/324133.324234.

[15] David Chase and Yossi Lev. "Dynamic circular work-stealing deque". In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 2005, pp. 21–28.

[16] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. "Optimising purely functional GPU programs". In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[17] Robert Clifton-Everest, Trevor L McDonell, Manuel MT Chakravarty, and Gabriele Keller. "Embedding foreign code". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2014, pp. 136–151.

[18] Trevor L McDonell, Manuel MT Chakravarty, Vinod Grover, and Ryan R Newton. "Type-safe runtime code generation: accelerate to LLVM". In: *ACM SIGPLAN Notices* 50.12 (2015), pp. 201–212.

[19] Robert Clifton-Everest, Trevor L McDonell, Manuel MT Chakravarty, and Gabriele Keller. "Streaming irregular arrays". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 2017, pp. 174–185.

[20] H. Partsch and R. Steinbrüggen. "Program Transformation Systems". In: *ACM Comput. Surv.* 15.3 (Sept. 1983), pp. 199–236. ISSN: 0360-0300. DOI: 10.1145/356914.356917. URL: https://doi.org/10.1145/356914.356917.

[21] Eelco Visser. "A Survey of Rewriting Strategies in Program Transformation Systems". In: *Electronic Notes in Theoretical Computer Science* 57 (2001). WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming, pp. 109–143. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00270-1. URL: https://www.sciencedirect.com/science/article/pii/S1571066104002701.

[22] Martin S Feather. "A system for assisting program transformation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.1 (1982), pp. 1–20.

[23] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1 (2008). Special Issue on Second issue of experimental software and toolkits (EST), pp. 52–70. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.11.003. URL: https://www.sciencedirect.com/science/article/pii/S0167642308000452.

[24] Hsiang-Shang Ko and Jeremy Gibbons. "Programming with ornaments". In: *Journal of Functional Programming* 27 (2016).

[25] Filip Srajer, Zuzana Kukelova, and Andrew W. Fitzgibbon. "A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning". In: *CoRR* abs/1807.10129 (2018). arXiv: 1807.10129. URL: http://arxiv.org/abs/1807.10129.