Utrecht University

MASTER'S THESIS

Fusion of Expand and Permute in Accelerate

Author:
Jaan J. G. Van Gils
6200443
j.j.g.vangils@students.uu.nl

Program:
Computing Science

Track:
Programming Technology

Supervisors:

Ivo Gabe de Wolff Msc
Prof. Dr. Gabriele K. Keller
Dr. Wouter S. Swierstra
Dr. Trevor L. McDonell

July 7, 2023

Abstract

Data-parallel array languages, like Accelerate, provide data-parallel operations as high-level functions for which no low-level programming mastery is required. Permutation and flattening by expansion are two of such operations that are useful for handling irregular nested data. Accelerate transforms the high-level code written by the user to low-level code. Because many programs are inherently memory-bound, it is beneficial to minimise the number of repeated memory loads and the number of temporary, intermediate arrays generated by this process. In this thesis, we study whether the number of temporary arrays can be reduced when performing a permutation after an expansion. Moreover, we will study the performance benefits of such a reduction.

Contents

1 Introduction

Declarative, combinator-based languages like Accelerate [1], Futhark [2], Nikola [3] and Obsidian [4] specialise on performing data-parallel array operations. They provide a higher-level interface for expressing parallel computations over arrays, so that programmers are not required to have extensive low-level knowledge to execute parallel operations efficiently on hardware. This is facilitated by the combinator-based nature of these languages, as complex operations can be created by combining multiple primitive ones.

Accelerate works by generating low-level machine code, that corresponds with the high-level Accelerate code that is written. It provides the option to generate machine code for various architectures. The specifics of this code generation are discussed in further detail in section 2.4.

To further optimise code, Accelerate incorporates an optimisation known as array fusion [5]. Fusion is a code transformation that combines operations, so that less sequential operations are needed during the execution of parallel code. This is discussed in more detail in in section 2.3.

Two particular operations are permutation and flattening by expansion [6]. These operations are discussed in the respective sections 1.1 and 1.3. Permutation and flattening by expansion play an important role in the parallel array-programming model, as they are useful in many different algorithms. As far as we know, the fusion of these two operations has not yet been explored in the context of functional array languages. The aim of this thesis is to develop and evaluate a technique for the fusion of the permutation and expansion operations.

We choose to analyse this by expanding a language that already supports fusion and permutation. The system of choice is Accelerate, which is a language embedded in Haskell [7]. It has the benefit that it provides a robust framework in which experimentation will be easy. To do the evaluation, we use the GPU backend of Accelerate.

1.1 Permutation

Sometimes, it is useful to rearrange values in an array. A useful operation for this is the *forward permutation* operation, also known as the *scatter* operation. This permutation is defined as an index mapping from a source index to a result index [8]. In Accelerate, the type of this operation is as follows:

```
permute ::
   (Exp a -> Exp a -> Exp a)        -- Combination function
   -> Acc (Array sh' a)             -- Array of default values
   -> (Exp sh -> Exp (Maybe sh'))   -- Index permutation
   -> Acc (Array sh  a)             -- Array of source values
   -> Acc (Array sh' a)             -- Array after permutation
```

Permute works by moving elements from the array of source values to a copy of an array of default values[1]. To accommodate this, `permute` takes the following four arguments:

1. The combination function, that defines how to combine the element in the defaults array[2] with the source element that is permuted to the same index.
2. The array of default values, that contains default values for the result. This is necessary, because the permutation may not be surjective. That is, the index permutations may not cover all possible target indices.
3. The index permutation function, which is the index mapping that defines the target index for every source index. Not all source indices have to map to a target, therefore the result is a `Maybe`. Elements that map to `Nothing` are discarded. There are no restrictions on the used mapping. It may be partial, non-surjective and/or non-injective.
4. The array of source values, that defines the elements that are permuted into the resulting array.

Since the index permutation function may not be injective, it is possible that one element in the result array depends on multiple elements from the source array.

Figure 1 gives an overview of how the different arguments cooperate to build the resulting array. In this figure, a permutation using index mapping p and combination function $\oplus$ of values from the source array $\mathbf{v}$ to defaults array $\mathbf{d}$ is visualised. In this example, source index 0 is mapped to target index 0, source indices 1 and 2 are both mapped to target index 1 and source index 2 is mapped to `Nothing`.

As only one element is mapped to target index 0, the result is $\oplus$ applied to v_1 and d_1. Because two elements are mapped to index 1, the result becomes $\oplus$ applied to the default value v_3, v_2 and d_2. This means that to assure that the result is deterministic, $\oplus$ must be associative and commutative if it is not injective. As nothing is mapped to target index 2, the value remains the default value d_3.

1.2 Segmented Scan

A *segmented scan* is a very useful operation to perform multiple scans within one array. Specifically, the scans are performed on different, disjoint parts/segments of the array. In Accelerate, the type of a (left to right) segmented scan is as follows:

[1]Sometimes the array of defaults is not used anywhere else. In this case, the elements can be moved to the original array instead of to a copy.

[2]This is not necessarily the element that was originally in the defaults array. If multiple elements are permuted to the same index, it can be another source element that was already permuted.

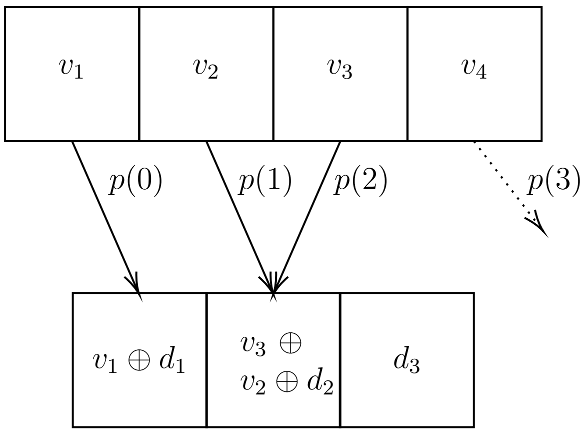


Figure 1: Example of permutation into an array of default values d_i, using index mapping p and combination function $\oplus$.

```
scanlSeg ::
  (Exp e -> Exp e -> Exp e)      -- Combination function
  -> Exp e                       -- Initial value
  -> Acc (Array (sh :. Int) e)  -- Array of source values
  -> Acc (Vector Int)            -- Array of segment lengths
  -> Acc (Array (sh :. Int) e)  -- Array after segmented scan
```

Note that the combination function should be associative, so that the result of the scan can be computed in parallel. An example of where a segmented scan can be useful is calculating multiple prefix sums within one array. This is shown in the following example:

```
>>> let source = use $ fromList (Z:.10) [1,1..] :: Acc (Vector Int)
>>> let segments = use $ fromList (Z:.4) [1, 2, 3, 4]
>>> run $ scanlSeg (+) 0 source segments)
Vector (Z :. 14) [0,1,0,1,2,0,1,2,3,0,1,2,3,4]
```

The result indeed consists of the prefix sums of the segments in the `source` array. All prefix sums for all segments start with the initial value that we set to 0. Note that as `scan` prepends the initial value to the front of the result, the final result has four more elements than the source array.

1.3 Flattening by Expansion

The `expand` operation, as introduced by Elsman *et al.* [6], is useful for flattening a certain form of nested data where the inner data structures may have different sizes. The `expand` operation can do this flattening while masking some of the low-level abstractions. Efficient flattening of nested parallelism is important, as flattening can sometimes lead to unnecessary overhead [9]. The form of irregular nested parallelism that can be flattened by expansion, is one where all data in a nested collection can be accessed using a one-dimensional index. An example of this form of irregular nested parallelism, can be found in figure 2. In this example, the array **e** contains arrays of various sizes. The data in any array $\mathbf{v_i}$ can be accessed using regular array indexing.

Elsman *et al.* [6] defined the `expand` operation to have the following type[3]:

```
expand :: (a -> Exp Int)
       -> (a -> Exp Int -> b)
       -> Acc (Vector a)
       -> Acc (Vector b)
```

The `expand` operation flattens the `input` array to a target array. To do this, it uses three arguments.

1. The `size` function that specifies how many target elements every element of the input array will expand to.
2. The `get` function that takes an element `a` from the input array and an index `i` $\in \{0, \ldots, ($`size a`$) - 1\}$, and returns the target element that will be at position `i` of the expansion of `a`.
3. The `input` array, which is the array of source values that are used by the above functions.

Consider the following example.

```
expand (\a -> 3) (\a i -> a * i) [1, 2, 3, 4]
```

In this example, every element has three target elements. For an element in the input, the target element at index `i` will be the original element multiplied by `i`. Which results in the following.

```
[0, 1, 2, 0, 2, 4, 0, 3, 6, 0, 4, 8]
```

1.3.1 Implementation

Flattening by expansion can be defined using basic data-parallel primitives and therefore can be directly translated to Accelerate. A possible implementation in Accelerate is given in listing 1.

Both `idxs` and `iotas` are temporary arrays, used to create the result array. The `idxs` array dictates the source index that an element in the result array comes from. That is, the element at index i of the `idxs` array defines the source index for the element at index i of the result. The `iotas` array stores an index that is combined with an element in the `get` function. Concretely, in the example from the previous section the two arrays would be as follows:

```
idxs  = [0,0,0,1,1,1,2,2,2]
iotas = [0,1,2,0,1,2,0,1,2]
```

[3]This is the corresponding type in Accelerate, ignoring type class constraints.

```haskell
segmScan
    :: (Exp b -> Exp b -> Exp b) -> Exp b
    -> Acc (Vector Bool) -> Acc (Vector b) -> Acc (Vector b)
segmScan op ne flags as = map snd scan
    where zp    = zip flags as
          scan =
            postscanl (\(T2 x_flag x) (T2 y_flag y)
                -> T2 ( x_flag || y_flag) (if (y_flag) then y
                    else (x `op` y))) (T2 False_ ne) $ zp

replIota :: Acc (Vector Int)
          -> (Acc (Vector Int), Acc (Vector Bool))
replIota reps =
    let s1    = scanl (+) 0 reps
        fld   = s1 !! (length s1 - 1)
        tmp   = scatter s1 (fill (I1 fld) 0)
                $ (enumFromN (shape reps) 0)
        flags = map (>0) tmp
    in  (segmScan (+) 0 flags tmp, flags)

segmIota :: Acc (Vector Bool) -> Acc (Vector Int)
segmIota flags = map (\x -> x-1)
    $ segmScan (+) 0 flags (fill (shape flags) 1)

expand size get arr =
    let sizes          = map size arr
        (idxs, flags)  = replIota sizes
        iotas          = segmIota flags
    in zipWith (\i j -> get (arr !! i) j) idxs iotas
```

Listing 1: Implementation of expand in Accelerate

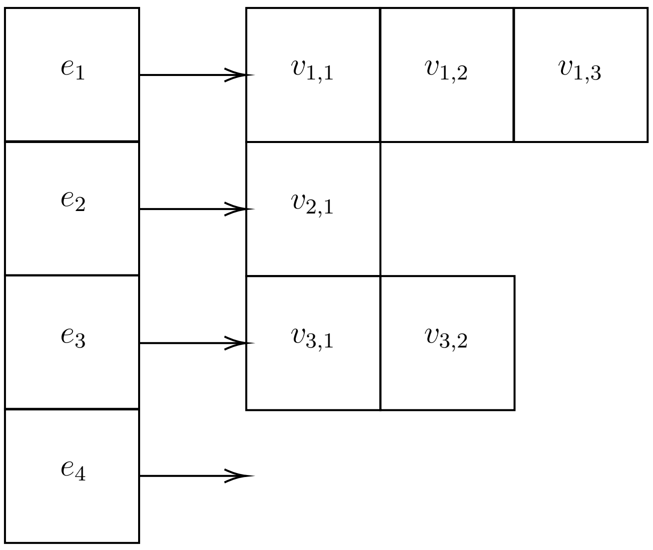


Figure 2: Example of irregular nested data, where the flat vector **e** generates the arrays $\mathbf{v_1}$, $\mathbf{v_2}$ and $\mathbf{v_3}$. Note that nested data can also be of length 0, as shown by e_4.

1.4 Permute after Expand

In many use cases of `expand`, permutation operations appear directly after flattening by expansion. In fact, half of the examples from the paper introducing `expand` [10] perform a permutation after an expansion. Examples of this are finding primes using the Sieve of Eratosthenes and rendering a 3D terrain. They even note the fusion of permutation with expansion as a possibility for future work. Another example of a permutation occurring after an expansion is updating distances in a shortest path algorithm. As noted in the previous section, this currently requires multiple kernels in Accelerate.

Looking at the purpose of the temporary arrays `idxs` and `iotas` in the implementation of the `expand` operation leads us to believe that while the `expand` operation is already data-parallel, optimisations may be possible in certain cases. Namely, if we permute directly after expanding, the scans may be redundant. When an expanded array is permuted instantly, we could directly generate an index and then write the element to its position in the new array. This would take away the overhead of generating three intermediate arrays: `idxs`, `iotas` and the result of `expand`. This thesis studies whether this leads to speedups.

1.5 Research Question

Research question. Does the fusion of expansion and permutation offer an increase in performance?

Previous work on fusion in the context of different kernels has shown that fusion can lead to reduced overhead. We study if this is also possible when performing a permutation after flattening by expansion. Our chosen metric for performance is the run time.

1.6 Contributions

We extend the work of Elsman *et al.* [6], by providing a way to reduce some of the overhead that is introduced by instantiating intermediate arrays when a permutation is performed directly after an expansion. Moreover, we discuss different ways in which the code generation for such a kernel can be implemented. The effectiveness of the discussed kernels is evaluated, and possible reasons for the results are given. Concretely, the contributions of this thesis are the following:

1. A custom method for the fusion of the `expand` and `permute` operations is described, and implemented in Accelerate.
2. A method to generate low-level code for the fusion transformation of `expand` with `permute` is discussed and implemented.
3. Benchmarks assessing the effectiveness of the fusion of `expand` with `permute` are performed.

2 Preliminaries

2.1 An Embedded Language

Programming languages often support many different mechanisms like classes, modules and higher-order functions. When developing a particular application, this high level of abstraction is not strictly necessary. The only mechanisms necessary are those that are useful for the domain that the application is targeted at. For this reason, *domain-specific languages* (DSLs) [11] are developed. A DSL provides the user with basic constructs that are useful for a certain domain. A very basic example is the Excel formula language, targeted at the domain of mathematics.

A programming language that is written within a different host language is called an *embedded language*. Such a language is written using the generic mechanisms provided by the host language. There are two embedding techniques [12, 13]. In a *shallow embedding*, terms in the embedded language are translated to the semantics of the host language. Evaluation of terms is often done by the host language. In a *deep embedding*, terms in the embedded language construct a representation of a computation. This representation then needs to be traversed for the evaluation. This evaluation is usually done at run-time.

Accelerate is a deeply embedded, domain-specific language. Since it is deeply embedded, it stores the representation of a parallel computation in a custom data type. This representation is called an *abstract syntax tree*. Concretely, it stores this in the `Acc` data type. The expressions that are used to form sequential computations are also deeply embedded. They have the type `Exp t`, where `t` is the result type of the expression. Both of these data types are *Generalised Algebraic Data Types* (see section 2.4). This helps to ensure type safety within the embedded language.

The parallel computations in Accelerate work on arrays. An array has the type `Array sh t`. It has a shape `sh` that describes how many and what form of dimensions it has. The type variable `t` defines what the type of the elements in the array is.

An example of this embedded representation can be seen when we look at `map` in Accelerate. We provide the Haskell type of `map` as a reference, denoting it as `Prelude.map`. The types of `Prelude.map` and `map` in Accelerate are as follows.

```
Prelude.map :: (a -> b)
            -> [a]
            -> [b]
```

```
map :: (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)
```

In Accelerate, the evaluation of computations happens at run-time. The `run` function can be used to perform this evaluation. This function has the following type:

```haskell
run :: Arrays a
    => Acc a
    -> a
```

The following is an example of the difference between the evaluation of `Prelude.map` and `map`. In the example, we compute the squares of the numbers 1, 2, 3 and 4.

```haskell
>>> let xs = [1, 2, 3, 4]
>>> Prelude.map (\e -> e * e) xs
    [1,4,9,16]
>>> map (\e -> e * e) (use $ fromList (Z:.4) xs)
    map (\x0 -> x0 * x0) (use (Vector (Z :. 4) [1,2,3,4]))
>>> run $ map (\e -> e * e) (use $ fromList (Z:.4) xs)
    Vector (Z :. 4) [1,4,9,16]
```

The type of `map ...` is `Acc (Vector Int)`. We see that it is indeed the representation of a computation, as it outputs the abstract syntax tree (AST) that is generated by Accelerate. The type of `run $ map ...` is `Vector Int`, thus the evaluation of the aforementioned computation.

2.2 GPUs

In the very beginning, computations on a computer used to be performed by one component: the central processing unit (CPU). This component is set up to be versatile; capable of performing many different tasks. Since then, new hardware components have been introduced that are focused on achieving better performance for a specific task. An example of a component that specialises on a task is the graphics processing unit (GPU). It was introduced as a more efficient way to render graphics. Due to developments like the release of the CUDA API [14] for NVIDIA graphics cards, GPUs are utilised in more generic computations as well. Accelerate is also able to use the GPU to perform its array computations. A GPU is able to achieve this specialisation, because its chip layout differs from that of a CPU. Figure 3 shows such a difference in layout. In this figure we see that a GPU has many more compute cores, and dedicates less chip space to caches and controllers. It can thus process more data at the same time, but at the expense of fast-to-access memory.

2.2.1 NVIDIA GPU Architecture

A GPU is focused on providing a higher instruction throughput and a higher memory bandwidth than a CPU. This is possible because the GPU hardware is specifically designed for highly parallel computations. A GPU thus tries to perform as many operations in parallel as possible. This is realised by devoting more transistors to processing instead of devoting them to data caching and flow control, as can be seen in the schematic in figure 3. By devoting more transistors to

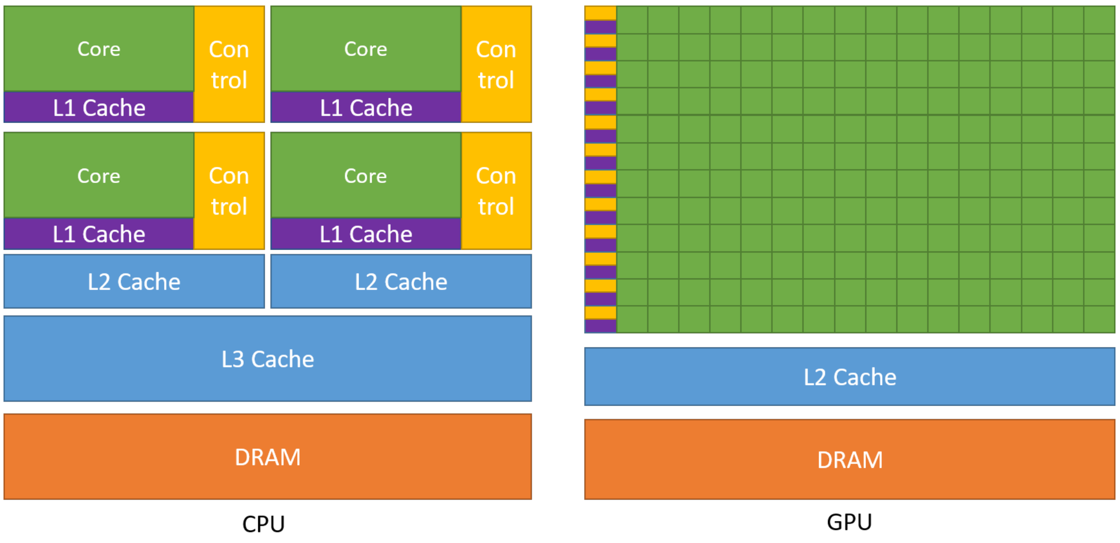


Figure 3: Distribution of resources on a multicore CPU chip versus distribution of resources on a GPU chip. Image from *CUDA C++ Programming Guide* [15].

processing, the latency of memory accesses can be hidden by performing computations of threads that do not need data from memory (yet). These computations can be performed while other threads await their requested memory access.

2.2.2 Kernels

To perform computations on the GPU, the programmer has to write a special GPU program. In OpenCL [16] and CUDA, such a program is called a *kernel* [15, 17]. A kernel is a Single Program Multiple Data (SPMD) program. This means that the same program is executed on multiple sets of data.

2.2.3 Threads

The computations on a GPU are performed by threads. The threads are divided into groups of 32 threads called *warps*. These warps are then grouped into so-called *thread blocks*. The amount of threads in a thread block depends on the amount of resources required by a thread, however there is an upper limit of 1024 threads[4], thus 32 warps per block. Finally, the blocks are divided over multiple multithreaded Streaming Multiprocessors (SMs). The collection of all thread blocks is called a grid.

The SMs contain the actual hardware for running threads. Figure 3 shows the general layout of a SM. In this figure, we see that a SM consists of compute cores (green). Moreover, it has two types of memory (purple) and some schedulers (yellow). A SM also has load/store units that handle global memory access. These

[4]This is the upper limit for compute capability 9.0.

resources are shared by all blocks running on the SM. Every row of cores, L1 caches and controllers is shared by a warp. This is why the amount of blocks that can run on one SM depends on the amount of memory and registers required by the assigned thread blocks. The SMs employ a Single-Instruction, Multiple-Thread (SIMT) structure. This is done by grouping threads from a warp that are able to execute into SIMT units. In this way, all threads that want to perform the same instruction can run this instruction simultaneously. This is possible because of *Independent Thread Scheduling* [15], where each thread keeps a program counter and call stack. In architectures before Volta, this worked differently. Prior to Volta, one program counter was kept per warp, and that instruction would be executed in all threads that were on that same instruction.

2.2.4 Memory

A GPU has several basic types of memory. The first is global memory. This is available to all threads on all SMs. This memory is not on a multiprocessor chip, and therefore is the slowest to access. It can be used to share data between threads that are not part of the same block. Shared memory is available on the chip, and is shared by all threads in the same block. It can be used to do basic communication between threads in a block. Every thread in a block is able to read from and write to it. Finally, there are local memory and registers. These are only accessible by one thread. They can be used to temporarily store variables that a thread needs later on in its execution. This hierarchy of GPU memory is also shown in figure 4.

2.2.5 Synchronisation

For some programs, it may be necessary that all threads have run all instructions up to a certain point. Because threads run independently of each other, they are not guaranteed to be at the same place in the program at the same time. In order to ensure that all threads have executed certain instructions, it is possible to synchronise threads using barriers. When a thread hits such a barrier, it waits until all other threads have also reached this barrier. There are different levels of synchronisation. Threads can be synchronised within a warp, within a thread block, or globally within the grid. Synchronisation can lead to deadlocks when one of the threads is not able to hit the barrier.

Synchronisation is also needed to guarantee that all threads agree on the state of shared and global memory. This is because there is no implicit thread synchronisation when memory is accessed, thus barrier synchronisation should be used if it is important that the state of the memory is the same for all threads. For example, when the `permute` operation combines a value with the value present in the array of defaults and writes it to memory, all threads should read the new (combined) value.

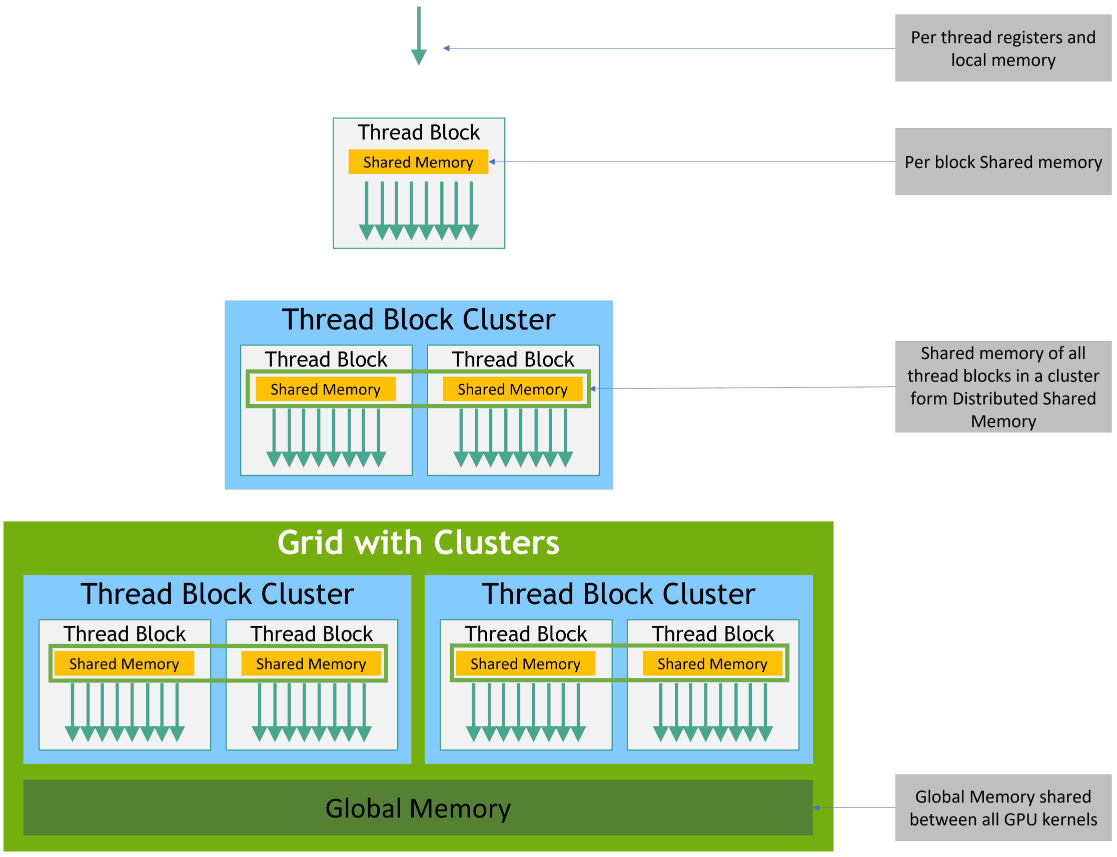

Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Global Memory
Global Memory shared between all GPU kernels

Figure 4: Memory hierarchy of a GPU. Shows at what level a given type of memory is available. Image from *CUDA C++ Programming Guide* [15].

2.2.6 Compute Capability

The capabilities of GPUs have changed over the years. For example, specifications like the available memory have changed. The capabilities of a GPU are indicated by its *compute capability*. This number is like a version number given to the architecture. At the time of writing, the latest compute capability is 9.0. It is important to take this compute capability into account, because some optimisations may not be possible on certain GPUs. For example, it is not possible to use *warp shuffling* [18] on a GPU with a capability below 3.0. Furthermore, programs may behave differently when ran on a GPU that targets a different compute capability. An example of this is implicit thread synchronisation within a warp, which was no longer available from capability 7.0 (the Volta architecture) [15] onward.

2.3 Fusion

To make programs run faster, they need to be optimised. This can be done manually by the programmer, or automatically by the compiler. When the programmer optimises by hand, they are able to use certain information about the program (e.g. invariants) to make functions more efficient. On the other hand, the compiler might recognise some patterns used by the programmer, and transform them into more efficient code.

A number of compiler optimisations combine operations. By combining operations, some of the overhead for storing and reading instructions can be avoided. Early examples of this are super-combinators [19] and superoperators [20].

Another way to optimise a program is by combining loops. When the compiler sees two adjacent loops, that loop over the same range, it can choose to move the loop bodies into a single loop body if there is no data-dependence between the loop bodies. This transformation is known as *loop fusion* [21]. The advantages of loop fusion are that:

- The overhead of keeping track of the loop condition is reduced
- The code space is reduced
- More instructions are exposed for parallel execution and for local optimisation

Loop fusion is a so-called *source-to-source* [22] transformation. This means that the source code is optimised by slightly modifying it, rather than by introducing new primitive operations for the language that the compiler compiles to.

2.3.1 List fusion

In the functional programming model, programs are split into multiple smaller operations. An example of this is shown in listing 2. In this example, the sum of squares is computed using multiple small steps. First, an array of integers from 1 to n is constructed. Next, the square is computed for all elements of this array. Finally, the `sum` function is used on the array containing the squared numbers.

Some overhead is introduced because all small operations produce an intermediate result. This is less of a problem in imperative languages, as the same algorithm can be expressed as a single for loop like the one in listing 3. In this program, the steps of computing squares and summing are both done in the same loop operation. There are no intermediate results because of this; the final result is computed by the main loop.

```
sum (map square (1 `upto` n))
```

Listing 2: Sum of squares in a functional language.

```
result = 0
for i in range (1, n):
    result += i * i
```

Listing 3: Sum of squares in an imperative language.

When running list combinators consecutively, as is done in the functional programming model, intermediate lists are generated. For the program in listing 2, three lists have to be constructed. One for the result of `upto 1 n`, one for the result of `map square` and one for the result of `sum`. Because it costs time to allocate, examine and deallocate the intermediate lists, it is beneficial if they can be omitted. In some cases, this can be done by transforming a functional program into a imperative one, where the imperative one is *listless* [23]. An example of this is the transformation of listing 2 into listing 3.

Another possibility is to perform *deforestation* [24]. In this source-to-source transformation, combinators are merged to form a recursive definition that reduces the number of required intermediate lists. The program in listing 4 is a deforested version of the program in listing 2. Over the years, deforestation has been refined [5, 24, 25]. An additional benefit of the method by Chakravarty and Keller [25], is that it it generalises to a parallel context. This makes it relevant for Accelerate. Fusion is also beneficial for machine learning applications, as it exposes a number of optimisation opportunities like fewer scans of input data [26].

```
h 0 1 n
where
h a m n = if m > n
          then a
          else h (a + square m) (m + 1) n
```

Listing 4: Deforested version of the program in listing 2.

2.3.2 Array Fusion

One of the refinements of deforestation that combines operations and reduces the amount of memory needed for storing temporary, intermediate values is called *array fusion* [5]. This is a useful optimisation, as many programs are inherently memory-bound. Array fusion works as follows. In essence, a naive implementation of an array language generates at least one kernel for every operation. As every kernel generates an array containing its output[5], this can lead to a large overhead. A way to optimise this is by combining kernels. When kernels are combined, the temporary output array that would otherwise be generated after the first kernel, is handled implicitly in the combined kernel. This removes some of the memory access required, thereby speeding up code execution. This process of combining kernels is called *vertical fusion*.

Another way to reduce the number of memory accesses, is by reducing the amount of traversals over the same array. This can be achieved by combining the different traversals over the same array into a single traversal as with loop fusion. The process of combining traversals in this way is called *horizontal fusion*.

2.3.3 Fusion in Accelerate

In the current implementation of fusion in Accelerate, only vertical fusion is performed. To do this, array operations are separated into two categories [5]:

1. **Element-wise operations** (*Producers* in [5])
   Operations where each element of the result array depends on at most one element of the input array.

2. **Collective operations** (*Consumers* in [5])
   Operations where each element of the result array depends on multiple elements of the input array.

The fusion of kernels is then performed in two stages. First, sequences of element-wise operation are fused into one cumulative element-wise operation. This can be done using a source-to-source transformation on the AST. Then, element-wise operations that are followed by a collective operation are merged into the collective operation.

Element-wise operations can be rewritten to a special representation: one that keeps track of the size of an array, and stores a function that maps indices to corresponding values [5]. This is facilitated using the data structure given in listing 5.

The `Done` constructor embeds a manifest array into the AST. The `Yield` and `Step` constructors contain functions that form a structured traversal over an array. This is because `Yield` contains a function that returns an element given an index,

[5]This output array may be used as the input array for the following kernel.

```
data DelayedAcc a where
  Done  :: Acc a
        -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
        => Exp sh
        -> Fun (sh -> e)
        -> DelayedAcc (Array sh e)
  Step  :: (Shape sh, Shape sh', Elt e, Elt e')
        => Exp sh'
        -> Fun (sh' -> sh)
        -> Fun (e -> e')
        -> Idx (Array sh e)
        -> DelayedAcc (Array sh' e')
```
Listing 5: Special representation of element-wise operations, to aid fusion.

and `Step` contains an index- and value-space transformation, that can be applied to the array that the given index points to.

In the `DelayedAcc` data structure, subsequent element-wise operations can then further build on the mapping function of the previous element-wise operations. Collective operations need to stay intact however, because it is necessary to know how computations depend on each other to generate the most efficient implementation of a collective operation. This is why it is only possible to do element-wise/element-wise and collective/element-wise fusion and why it is not possible to do collective/collective fusion.

2.3.4 Fusion of Permute with Expand

As seen in section 1.3.1, `replIota` and `segmIota` are implemented in terms of a *segmented scan*. The same thus goes for `expand`, which is defined in terms of the aforementioned operations. Both segmented scans and permutations are seen as collective operations [5]. Fusion of these two operations is therefore not possible using the existing algorithm for fusion in Accelerate, as it does not support collective/collective fusion. Since `expand` is implemented using a segmented scan, it is currently not possible to fuse `expand` with `permute`.

2.4 Code Generation

The code generation in Accelerate is based around a concept known as *algorithmic skeletons* [27]. An algorithmic skeleton is the computational skeleton of an algorithm, where problem-specific details can be plugged in later. To the programmer, a skeleton is represented as a higher-order function[6] in functional languages, or a program "template" in an imperative language. The applied skeleton can then be compiled to its most efficient implementation for a given target language.

Algorithmic skeletons work because of the observation that any operation (e.g. `map`), will always be implemented using similar code, but will use different values for its arguments. It is thus possible to write a template for what an operation should look like, where only the specific values corresponding with the arguments should be filled in.

In Accelerate, the array operations are regarded as skeletons [28]. A code generator will then instantiate the implementation of those array operation-skeletons, so that runnable machine code is generated. This code generator is dynamic, so that code is generated at run-time. Because of this, it is possible to perform device-specific optimisations. Currently, Accelerate compiles to the LLVM [29] language, which is further compiled using the LLVM compiler to multi-core CPU code or GPU code.

Using the typed LLVM language as an intermediate language has the advantage that it helps to ensure compiler correctness [30]. To make full use of this, Generalised Algebraic Data Types (GADTs) [31] are used to define expression types. This way, the type of an Accelerate expression can be embedded into the Haskell type system. In GADTs, a data type contains a type variable that is not used by a constructor. The constructor can however instantiate a more explicit version of the data type. This way, basic type checking can be performed. See listing 6 for an example of a small expression language, defined using GADTs.

The code generation is the final step in the Accelerate pipeline. It thus happens after the fusion of operations. Because of this, we have to introduce a representation of fused operations in the AST, so that code generation is possible for fused operations.

```haskell
data Expr a where
    LitInt      :: Int  -> Expr Int
    LitBool     :: Bool -> Expr Bool
    IfThenElse  :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Listing 6: A small language of expressions, defined using GADTs.

2.5 Nested Data-Parallelism

Performing one operation in parallel over an array of data is called *data-parallelism* [32, 33]. Data-parallelism is very useful on GPUs, as they are designed to run one set of instructions on large amounts of data. There are two types of data-parallelism. In *flat data-parallelism* (FDP), the operation that is performed on every element of the data must be sequential.

[6]A higher-order function is a function that takes other functions as its arguments.

The second type of data-parallelism, is *nested data-parallelism* (NDP). In a language that supports nested data-parallelism, the function that is applied to every element of data can be any function, even a parallel one. In languages that support NDP it is thus possible to apply a parallel function multiple times in parallel [34]. A simple example of this is a list of lists. Take the list `xs` defined as `[[0,1,2], [2,1], [3]]`. This is a list containing lists of integers. We can then apply the `sum` function over all inner lists of `xs`. The result of this would be `[3, 3, 3]`.

In this example, the nested parallelism is as follows. The function `sum` is mapped over `xs`. This map can be performed as a parallel operation. The `sum` function can also be a parallel function, for example it can be implemented as a parallel fold. Thus, the parallel map launches a different parallel function. This can also be seen as a nested loop, where the `map` is the outer loop, and the `sum` is calculated by an inner loop.

When the length of the inner loop depends on the current element of the outer loop, we speak of *irregular nested parallelism* [34]. In the example of mapping `sum` over `xs`, the complexity of the sum depends on the amount of elements in the current list that is being mapped. Therefore, it is an example of *irregular nested parallelism*.

Nested (irregular) parallelism is useful in divide-and-conquer algorithms. In divide-and-conquer algorithms, the input is split into multiple smaller problems, each of which are then solved. The results of the smaller problems are then merged, to form the result of the original problem. In cases where the sub-problems can be solved in parallel and are solved using a parallel function, nested parallelism can be used to benefit from highly parallel execution. An example of this is the following algorithm for calculating a the sum of an array:

```
sum []      = 0
sum (x:[])  = x
sum xs      =
    let (firstHalf, secondHalf) = split xs
        firstSum  = sum firstHalf
        secondSum = sum secondHalf
    in  firstSum + secondSum
```

The function `split`, splits a given list into two halves. This is the 'divide'-step. Running the `sum` functions is the 'conquer'-step. Note that we can run `sum firstHalf` and `sum secondHalf` in parallel, as they are independent of each other. Moreover, sum itself is a parallel function, as it can perform this parallel computation. We thus conclude that `sum` is an example nested parallelism, because it executes parallel operations in parallel.

2.5.1 Flattening

Because the inner collections in irregular nested parallelism can have different sizes, it is hard to balance the workload [10]. One possible solution is to turn the nested structure into a flat structure, thus to turn the NDP program into a FDP program. This process is called *flattening*. Flattening is well understood in the context of regular data-parallelism [35]. It is still difficult to flatten irregular data-parallelism so efficiently that it competes with hand-optimised code.

GPUs favour regular data. This is why it is important to transform irregular data into regular data to improve efficiency. One way to do this is by flattening the irregular nested data, before performing operations on the data.

3 Language Design

To introduce the fusion of expansion with permutation, the front- and backend of Accelerate require changes. The front-end needs support for `Expand` as a node in the AST and the type of the `Permute` node has to be adapted for easier fusion with `Expand`. Moreover, a node that contains information about the fused `Permute` and `Expand` nodes has to be introduced into the AST. The changes to the AST will be discussed in section section 3.1. A new function should be exposed in the language that can be used to generate the new `Expand` node. Moreover, the actual fusion of the `Permute` and `Expand` needs to be implemented in the existing algorithm for fusion. This is discussed in section 3.2.

In order to evaluate the execution, a kernel needs to be generated in the Accelerate backend. In section 4.1 we will propose a number of possible skeletons, together with their strengths and weaknesses.

3.1 Primitives

The first step in adding code generation, is to extend the AST. This is necessary, because code generation only happens for primitives in the AST. We start with a constructor for expansion. In this constructor, we need to store all relevant information for the expansion, so that it can be used later on. This is the basic information, like the `size` function, the `get` function, and the source array. Moreover, we want to store the result type of the expansion. The concrete constructor is found below in listing 7.

We also need to add a primitive that represents the fusion of expansion with permutation, as we want to generate code for it. This primitive will only be used internally, like the `Transform` primitive, and will not be exposed to the user. In it, we store the information of the expansion that is fused, as well as the information of the permutation that it is fused with. We do not need to store the source array of the permutation, as this is embedded within the data we store for the expansion. In listing 7, we see that `PermutedExpand` is indeed a combination of the `Permute` and `Expand` primitives.

Finally, the `Permute` primitive is changed, to better accommodate the fusion. The `Permute` primitive used to receive a combination function, an array of default values, an index permutation function and an array of source values. When fusing the `permute` operation with the `expand` operation, the `expand` operation should provide the permutation indices as part of the result of the `expand` operation. We a number of options:

1. Traverse the result of the expansion, to create a permutation function.
2. Make the permutation function get the indices from the result of the expansion directly.

3. Change the `Permute` primitive, so that it takes an array of permutation indices.

4. Change the `Permute` primitive, so that it takes an array of source values, combined with their permutation indices.

The first option is not ideal, as it would require an extra pass over the source array. This could add an $O(n)$ overhead. The second option and third options are also not ideal, because that would mean that the result of the expansion has to be split into an array of permutation indices and an array of source values. This would introduce a `let`-binding of the form `let e = (expand ...) in permute ... (map fst e) (map snd e)`. This would make the result infusible. Therefore, the last option is the best. This is because we can easily combine an array of source values and an index permutation values, into a combined array of source values and permutation indices. The concrete changes to the AST are also found in listing 7 below.

As the AST has been changed, functions that pattern match on it need to be adapted so that they also match on the newly introduced primitives. Moreover, all occurrences of the `Permute` primitive have to be updated, so that they use the correct constructor again.

3.2 Fusion

In the fusion algorithm, we need to introduce some method that fuses the `Permute` and `Expand` kernels. We choose to implement this as an extra pass over the tree, as not to interfere with the current fusion that is based on collective and element-wise operations. In essence, the new function will recurse through the AST. For specific primitives, it will fuse them with `Expand`. The extra pass is implemented as the `fuseExpand` function. A part of the implementation is given below in listing 8. The full set of changes can be found on GitHub[7].

Accelerate also contains a special primitive that represents a while-loop. This is the `Awhile` primitive. It contains a function that is the loop condition, a function that is the loop body and an initial value. At every iteration, the loop condition is checked. If it returns `True`, the loop body is run. If it returns `False`, the loop stops. The loop body is a function that takes one argument. The value that is initially passed is the initial value. The value that is passed to the loop body in iteration $i + 1$ is the result that is returned by the loop body in iteration i. At the end of the loop, the value that `awhile` returns is the value returned by the loop body in the last iteration.

The loop body also contains array computations. This means that during a fusion pass, the loop body must also be traversed to see if it contains any primitives that can be fused. The fusion algorithm thus needs to recurse on this function. We

[7]`https://github.com/goedestudent/accelerate`

```haskell
data PreOpenAcc (acc :: Type -> Type -> Type) aenv a where

  Expand ::
    TypeR e'
    -> Fun           aenv (e -> Int)        -- size function
    -> Fun           aenv (e -> Int -> e') -- element creation
                                            -- function
    -> acc           aenv (Vector e)        -- source array
    -> PreOpenAcc acc aenv (Vector e')

  PermutedExpand ::
    -- Data for the expand
    TypeR e'
    -> Fun           aenv (e -> Int)    -- size function
    -> Fun           aenv (e -> Int -> (PrimMaybe sh', e'))
                                        -- element creation
                                        -- function
    -> acc           aenv (Vector e)    -- source array

    -- Data for the permute
    -> Fun           aenv (e' -> e' -> e') -- combination function
    -> acc           aenv (Array sh' e')   -- default values
    -> PreOpenAcc acc aenv (Array sh' e')

  Permute ::
    Fun              aenv (e -> e -> e) -- combination function
    -> acc           aenv (Array sh' e) -- default values
    -> acc           aenv (Array sh (PrimMaybe sh', e))
                                         -- source array
    -> PreOpenAcc acc aenv (Array sh' e)
```
Listing 7: Additions to the Accelerate AST.

also need to do this in our additional pass to fuse Expand, so that any applicable primitives in the body of the while are also fused with Expand. Concretely, this leads to the implementation found in listing 8. In this listing, the fuseExpandFun function is used to fuse expansions that occur within the loop body function.

```
fuseExpandFun :: OpenAfun aenv t -> OpenAfun aenv t
fuseExpandFun fun = case fun of
    Abody b    -> Abody (fuseExpand b)
    Alam lhs b -> Alam lhs (fuseExpandFun b)

fuseExpand ::  OpenAcc aenv arrs -> OpenAcc aenv arrs
fuseExpand acc@(OpenAcc pacc) = OpenAcc $ case pacc of
    -- Special cases for fusion with expand
    Permute f d a -> case fuseExpand a of
        OpenAcc (Expand (TupRpair _ e) size get arr)
           -> PermutedExpand e size get arr f d
        a' -> Permute f (fuseExpand d) a'
    Map tp f a -> case fuseExpand a of
        OpenAcc (Expand _ size get arr)
           -> Expand tp size (f `compose2` get) arr
        a' -> Map tp f a'


    -- Continuation of AST traversal
    Awhile p f a
         -> Awhile (fuseExpandFun p)
           $ (fuseExpandFun f) (fuseExpand a)
    Backpermute shr sh f a
         -> Backpermute shr sh f (fuseExpand a)
    [...]
```

Listing 8: The fusion of expansion with permutation and function mapping.

The fuseExpand function is then called as the first step in the fusion pipeline. This is to prevent Map primitives being transformed into their delayed representation, making it impossible to fuse them with the Expand primitive.

4 Code Generation

With the fusion of kernels in place, we now need to provide the code generator in the backend with a skeleton implementation for the `PermutedExpand` kernel. While this thesis is aimed specifically at generating and evaluating CUDA code, the discussed adaptations can also be applied to generate code for other multi-core target devices. The general strategy is discussed in section 4.1. This section will also discuss some strategies that can be used to divide the work. Some details regarding permutation are discussed in section 4.2. After this, an optimisation in the binary search algorithm is discussed in section 4.3. Finally, adaptations to the execution engine are discussed in section 4.4. An implementation of the discussed concepts is given in Accelerate. This implementation can be found on GitHub[8].

4.1 Kernel Generation

We start by giving a high-level overview of what the generated kernel for a fused expansion should look like. In essence, we have to perform a nested loop: the outer loop goes over all elements of the input, and the inner loop goes over the indices required for that element. The size of the inner loop can differ for every element, as the `size` function might return a different value for every element. We thus operate over irregular nested data. This is why it is hard to divide work over the available threads: the workload for an element is not known in advance. In pseudo-code, the kernel is as follows:

```
for element in input:
    for i2 in range(0, size(element)):
        index, value = get(element, i2)
        if isJust(index):
            write(index, value)
```

As this code is ran on multiple threads, we need to come up with a strategy to balance the workload between the threads. There are multiple ways in which this can be done. At one end of the spectrum, one thread can fully process one element of the input. At the other end is to have one element of the input cooperatively processed by many individual threads. This results in every thread generating one element of the expansion output, and permuting that result. In such a case, we would likely want some sort of inter-thread communication, to prevent the costly operation of reading from the input for every thread. We now describe a number of different kernels that correspond with different workload distributions, ranging between the aforementioned extremes. These distributions all have their own advantages and disadvantages, which will be discussed. The upcoming sections use some GPU-specific variables. The definitions of these variables can be found in table 1.

[8] `https://github.com/goedestudent/accelerate-llvm`

Variable	Description	Uniqueness
`lane_id`	Identifier of thread within the warp	Within 1 warp
`thread_idx`	Global identifier of the thread	Globally
`thread_bidx`	Identifier of thread within the block	Within 1 block
`warp_id`	Identifier of the warp	Within 1 block
`block_id`	Global identifier of the block	Globally
`num_size`	Number of warps in a block	N/a
`block_size`	Number of threads in a block	N/a
`num_blocks`	Number of thread blocks running this kernel	N/a
`num_threads`	Total number of threads running this kernel	N/a

Table 1: Variables available to GPU programs.

4.1.1 1-per-thread Division

In the simplest implementation, one thread processes one element of the input. In pseudo-code, every thread will run the following kernel. Note that `thread_idx` refers to the global thread index.

```
def 1_per_thread():
    # This for-loop assigns every index to only one thread
    start, end, step = (thread_idx, len(input), num_threads)
    for i in range(start, end, step):
        element = input[i]
        for i2 in range(0, size(element)):
            index, value = get(element, i2)
            if isJust(index):
                write(index, value)
```

The main advantage of this kernel is its simplicity. No special checks or allocation logic is required, apart from pairing every input element with one thread.

Disadvantages of this kernel include that the workload of threads could become heavily unbalanced, as every element in the input can be expanded to an arbitrary amount of values. Moreover, not all threads will be utilised for small inputs. For example, a machine with 1024 available threads would have half of them idling on an input of 500 elements, even though those 500 elements are potentially expanded to millions of values.

One way to mitigate this, is by having multiple threads work on the same element of the input. This can be done in various granularities, the simplest of which are block-level and warp-level, which we discuss in the next sections.

4.1.2 1-per-block Division

First, we consider a strategy where one element of the input is processed by one block. To divide the input elements at a block-level, the kernel looks as follows:

```
def 1_per_block():
    # This for-loop assigns every index to only one block
    for i in range(block_id, len(input), num_blocks):
        element = input[i]
        for i2 in range(thread_bidx, size(element), block_size):
            index, value = get(element, i2)
            if isJust(index):
                write(index, value)
```

In this kernel, every input element is processed by one thread block. Every thread in the block then performs one expansion and permutation. This kernel is less susceptible to unbalanced workloads, albeit by a small amount. All threads are likely to perform an (about) equal amount of work, given that they all perform an expansion and the following permutation. Imbalance can still arise when one input element expands to a greater number of elements than others. In such a case, one block will need to perform more operations than the others. Moreover, when all input elements expand to less elements than the block size, there are threads that will be idle.

Furthermore the kernel is inefficient, as every thread will read the input element. This results in a lot of memory accesses, which can become costly for bigger inputs. To mitigate this, we can share the value between threads that work on the same input element.

4.1.3 1-per-block Division With Shuffling

One way to share data between threads, is by shuffling data within warps. By shuffling data in a warp, every element needs to be read *block_size/warp_size* times instead of *block_size* times. Recall that a warp consists of 32 threads, thus we reduce the amount of global memory accesses by a factor of 32. This sharing is easily added, as can be seen in the following kernel.

```
def 1_per_block_with_shuffle():
    for i in range(block_id, len(input), num_blocks):
        if lane_id == 0:
            element = input[i]
        __sync_warp()
        element = __shfl_idx(element, 0)
        for i2 in range(thread_bidx, size(element), block_size):
            index, value = get(element, i2)
            if isJust(index):
                write(index, value)
```

This approach does not work old GPUs, as the shuffle operation was not always available. The shuffle operations became available with the Kepler architecture, which corresponds with compute capability 3.0.

4.1.4 1-per-block Division With Shared Memory

When using shuffles, an element in the input is still read more than once. With thread blocks of 1024 threads, every element is still read 32 times when the warp size is 32 threads. One possibility to remedy this is by using shared memory, as this is shared by the whole thread block. While using shared memory is slower than shuffling as it uses three operations as opposed to one [36, 37], this may be balanced out due to the reduced amount of total memory access required. Moreover, this is the method recommended by NVIDIA when multiple threads in a block use the same data from global memory [38]. Moreover, this is a better way when the shuffle operation is not available on a GPU.

Implementing the sharing of input elements via shared memory would require the following insignificant change to the kernel from the previous section. Note that because shared memory is allocated per thread block [38], every thread in a block can write to and read from index 0. An obvious drawback of this approach is that if the input element expands to a very small amount of elements (e.g. if we use the expand as a filter), the writing to and reading from shared memory only adds additional overhead. However, using one thread block to expand the elements would also be inefficient in such a case.

```
def 1_per_block_with_shared_memory():
    for i in range(block_id, len(input), num_blocks):
        # The if now only runs for one thread per block, instead
        # of one thread per warp
        if thread_bidx == 0:
            element = input[i]
            shared[0] = element
        __sync_threads()
        element = shared[0]
        for i2 in range(thread_bidx, size(element), block_size):
            index, value = get(element, i2)
            if isJust(index):
                write(index, value)
```

4.1.5 1-per-warp Division

Another approach to reduce the number of global reads when using shuffles is by spreading the input over warps. This too is a relatively small change to the kernel that performs multiple shuffles of the same element. A drawback of this approach is that an element is expanded by only 32 threads. If an element expands to many values, it will take longer to compute because there are fewer threads (i.e. 32 instead of 1024) available to work on it concurrently compared to the situation where the input is spread over thread blocks.

```
def 1-per-warp():
    # Loop using warps instead of blocks
    for i in range(warp_id, len(input), num_warps):
        if lane_id == 0:
            element = input[i]
        __sync_warp()
        element = __shfl_idx(element, 0)
        for i2 in range(lane_id, size(element), warp_size):
            index, value = get(element, i2)
            if isJust(index):
                write(index, value)
```

4.1.6 *n*-per-block Division

Thus far, we have introduced various algorithms aimed at expanding one input element using one or more thread blocks. We can however also take the opposite approach. That is, we use one block to expand multiple input elements. This will be especially beneficial in situations where the source elements expand to a small number (less than the number of threads in a block) of elements.

In this approach, every block is assigned n input elements. Every block will then calculate how many target elements its assigned input elements are expanded to. Next, all blocks determine how to balance their workload over their available threads. This is done under the assumption that all expansions will take about the same time. Every block balances the expansions such that one expansion, that is running the `get` function, is done by one thread in the block.

The loop will then look as in listing 9. Note that there are certain important edge-cases that we need to take into account. In listing 9, `NUM_ELTS` refers to the number of elements n that one block will process. The calls that synchronise threads are necessary at their points in the listing, because we do not want threads to read old values from shared memory, or to overwrite values while other threads are still using them.

It is also important to note that in the loop that loads in the sizes, we need to take the case into account in which the input has a size that is not a multiple of n. If this is the case, the loop will start reading out-of-bound values. It must therefore be checked if `i + j` is still within bounds of the input array.

A Parallel Sum The prefix sum in listing 9 is calculated sequentially. This may impact the performance of the kernel, especially for larger n. We have therefore also implemented a version of the Sklansky prefix sum. The original algorithm from Grimshaw and Merrill [39] has been implemented in Haskell and has had its outer loop unrolled for further efficiency.

```
def n_per_block():
    for i in range(block_id * NUM_ELTS, len(input), num_blocks * NUM_ELTS):
        __sync_threads()
        # Load in all sizes
        for j in range (thread_bidx, NUM_ELTS, 1):
            value = input[i + j]
            smem_value[j] = value
            smem_size[j] = size(value)
        __sync_threads()


        # Calculate the prefix sum
        if thread_bidx == 0:
            for j in range (1, NUM_ELTS, 1):
                smem_size[j] += smem_size[j - 1]
        __sync_threads()

        total_elements = smem_size[NUM_ELTS - 1]
        for i2 in range(thread_bidx, total_elements, block_size):
            input_index = binary_search(i2, smem_size)
            element = smem_value[input_idx]
            get_index = 0 if input_index == 0
                          else smem_size[input_idx - 1]
            index, value = get(element, get_index)
            if isJust(index):
                write(index, value)
```

Listing 9: Implementation of the n-per-block kernel.

4.2 Permutation

In the previous subsection, we have discussed ways to expand the input in a parallel manner. We now present some important concepts that are needed for the permutation. First, recall that permutation requires a combination function $\oplus$. When we permute an expansion to the result array, we combine it with the value that was already in that location. That means that in the above kernels, the permutation is performed as follows.

```
def write(index, value):
    index = fromJust index
    output[index] = output[index] ⊕ value
```

This leads to the next problem: since all kernels run in parallel, it is possible that two kernels need to permute to the same index at the same time. They may then both read the same old value, and write their own new value. This is unwanted behaviour, because the read and writes should happen in one step. We want such a *read-modify-write* to happen atomically. There are two ways to implement this. The first is using CUDAs built in atomics. These are only available when the combination function is a primitive function like addition or multiplication. The second method is by using locks. This requires an auxiliary array in which it is indicated whether an element in the output is currently being read and modified (locked) or not. When we want to permute, we first acquire the lock. We then perform the read-modify-write and release the lock so that other threads can perform their own read-modify-writes.

4.3 Binary Search

In the n-per-block kernels we use binary search to find the input element that a given thread needs to work on. Therefore, the binary search should return the first element that is greater than or equal to the target value. This can be implemented in multiple ways. The simplest is using a simple loop as follows.

```
def binary_search(target, array):
    left, right = (0, len(array) - 1)
    while right > left:
        delta = left + right
        mid = (delta `quot` 2) + 1
        value = array[mid]
        if value < i:
            left, right = (mid, right)
        else:
            left, right = left, mid
```

This adds overhead, because the program now needs to keep track of the loop condition. This is unnecessary, as we know that the binary search will always perform $log_2(n)$ steps until the condition is true. We can thus unroll the loop, removing the need to check the condition at every step. We can do this efficiently if we first look for the last element that is smaller than the target value. We then know that the element after that must be the first element that is greater than or equal to the target value. Note that the `binary_search_unroller` function is *not* part of the kernel, but is used to generate the kernel.

```python
def binary_search_unroller(target, arr, bit, res):
    bit -= 1
    if bit == -1:
        return res
    mask = 1 << bit
    mid = res + mask
    if arr[mid] <= target:
        res = res | mask
    return binary_search_unroller(target, arr, bit, res)
```

```python
def binary_search(i, arr):
    n_bits = math.floor(math.log(len(arr), 2))
    target = i - 1
    res = binary_search_unroller(target, arr, n_bits, 0)
    if res == 0 and arr[0] > target:
        res =- 1
    return res + 1
```

4.4 Run-time Evaluation

Thus far we discussed the steps necessary to fuse the `expand` and `permute` operations and to generate code for the resulting fused kernel. The final step in making the kernel available to the user is providing the backend with the correct information to execute the kernel. In Accelerate, the kernel requires most of the information that permutation also requires. The execution function is therefore implemented as an adaptation of the execution function for forward permutation.

In essence, we start by creating array representations that can be used to allocate the necessary memory on the remote device. We then check if the defaults array can be modified in place. If it cannot be modified in-place, we create a copy of that array. If we need to run the kernel that requires an array of locks, we also allocate memory to store those locks. Finally, we copy the input array to the remote and execute the kernel using the allocated arrays.

5 Results

To test the effect of different methods for fusing permutation with expansion, several benchmarks have been ran to test their performance. We use micro-benchmarks, as well as some realistic use cases where permutation would be used after flattening by expansion. In this section we will discuss the benchmarks and the different results found after running the benchmarks. The benchmarks are run using the Criterion. This is a performance measurement framework for Haskell. Every benchmark is first executed 10 times as a warm-up. Performance is not yet measured during these runs. The benchmark is then run repeatedly for 5 seconds. The reported run time is the average over these runs. The benchmarks are run on a Linux machine with a *GeForce RTX 2080 Ti* graphics card, that has 12 gigabytes of VRAM. The system also has an *AMD Ryzen Threadripper 2950X 16-Core* processor and 64 gigabytes of system memory. This is less relevant for the benchmarks as they are targeted at measuring the GPU performance.

5.1 Micro-benchmarks

We perform a number of micro-benchmarks. They aim to test the performance of the different kernels in certain specific cases. They should provide an idea for the speedup that can be gained by using the proposed fusion in an algorithm of the same category as the benchmark. In these benchmarks, we take several factors into account that may influence the performance of the load balancing of the kernels. These factors are the *expand size*, the *permutation index distribution* and the use of *permutation as a filter*. In the following paragraphs, we will describe the rationale behind the concrete micro-benchmarks. The naming scheme of the benchmarks is as follows. The term $(p_1\%\ l_1 - u_1; \dots)$ denotes that with probability p_i, the size that a source element expands to comes from the uniform distribution $U(l_i, u_i)$. For the permutation targets, we use the following abbreviations:

- R: The index is chosen randomly from $U(0, \min(total\_elements, 1000))$.
- N: The index is `Nothing`, thus the result is not permuted to the result array.
- $r \cdot R + n \cdot N$: The index is R with probability r and N with probability n.
- U: The index will be unique, thus permutation never has to sequentialise two permutations.

(100% 0-10) $\mapsto$ R In this benchmark, a source element always expands to a small size. This is to give an insight into what kernel performs best in an algorithm where the expansion size is always small. The results are plotted in figure 5.

The first observation we can make using these results, is that the fused kernels are more than four times as fast as the unfused version. The next observation is

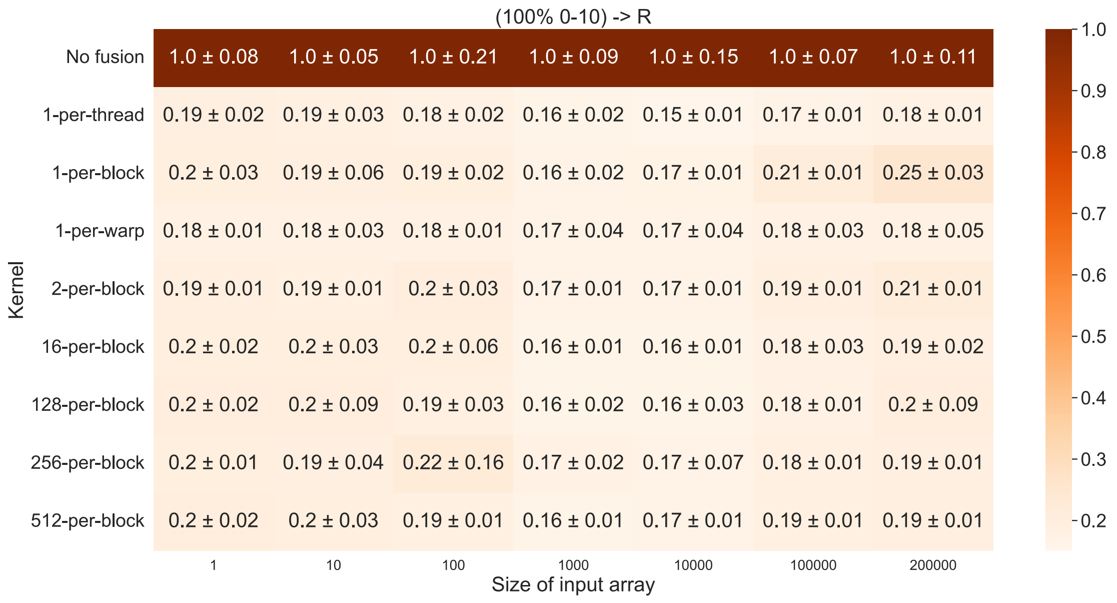
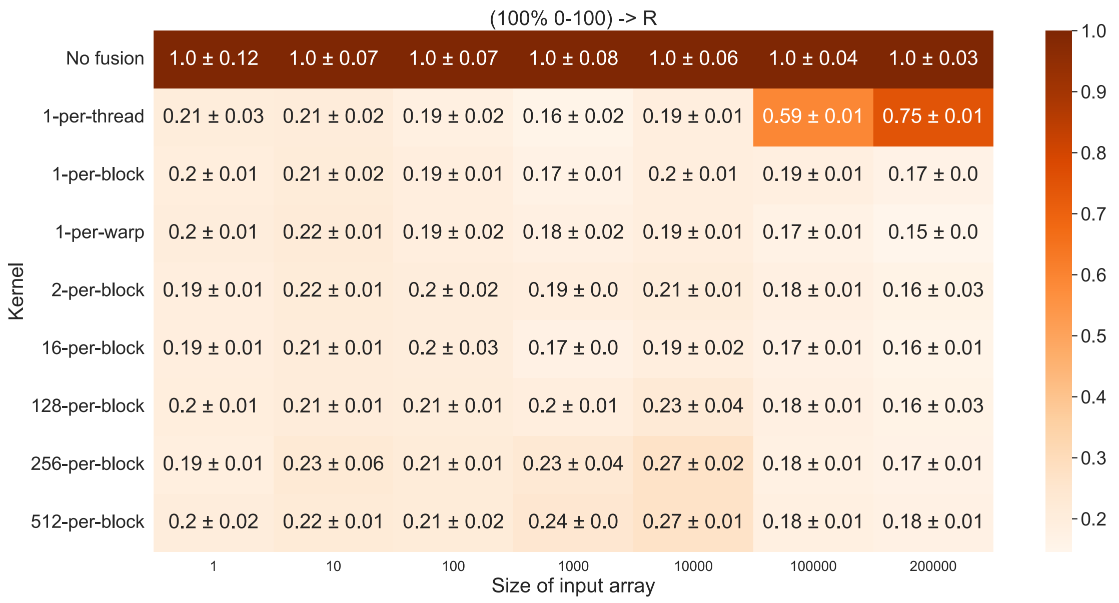


Figure 5: Average run time of kernels on the '(100% 0-10) -> R' benchmark, relative to the run time of the kernel without fusion.

that the n-per-block kernels are faster on the input of 200000 elements than the other kernels. Moreover, the n-per-block kernels are faster than the 1-per-block kernel for most inputs. This could be explained by the fact that all elements expand to a small size. Because a block has 1024 threads, it is very inefficient to process only one input element on one block. This is because at most 10 threads are utilised, in which case 1014 threads will not be utilised. Since an n-per-block kernel processes n times more elements than a 1-per-block kernel, it is able to do n times the amount of work that the 1-per-block kernel would be able to do. Note that this is only true if the amount of expansions done by the n-per-block kernel does not exceed 1024. Given that the expected expansion size for every element is 5, the ideal n would be $\lfloor 1024/5 \rfloor = 204$. We do indeed see that the 512-per-block kernel performs worse than the other n-per-block kernels.

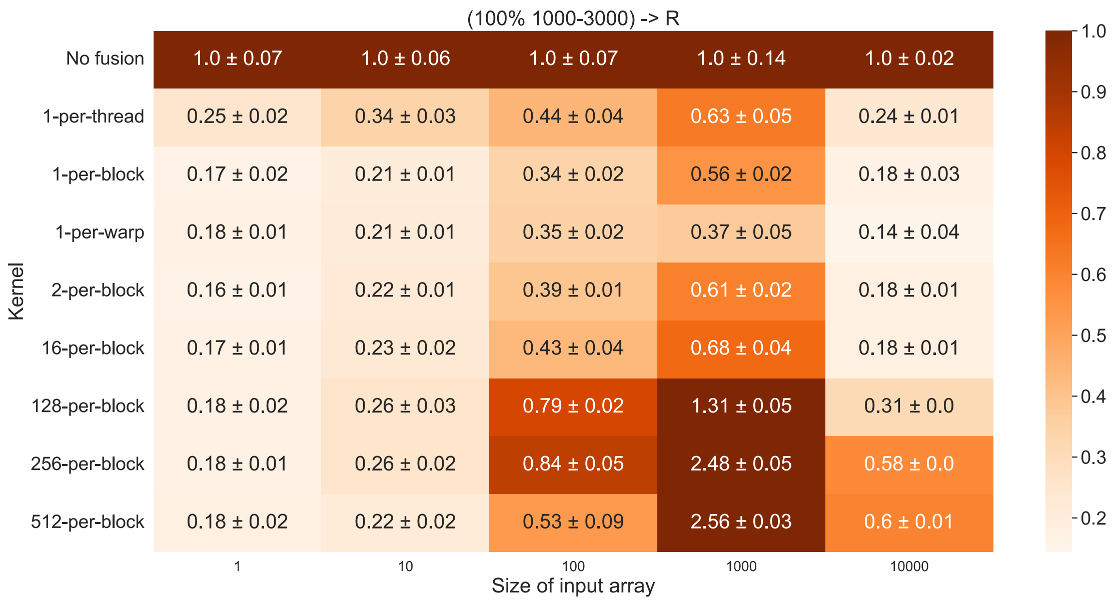
(100% 0-100) $\mapsto$ R In this benchmark, a source element always expands to a moderate size. This is to give an insight into what kernel performs best in an algorithm where the expansion size is moderately sized. The results are plotted in figure 6. In this figure, we again see that a speedup of around five times was achieved.

(100% 1000-3000) $\mapsto$ R In this benchmark, a source element always expands to a large size. This is to give an insight into what kernel performs best in an algorithm where the expansion size is always large. The results are plotted in figure 7.



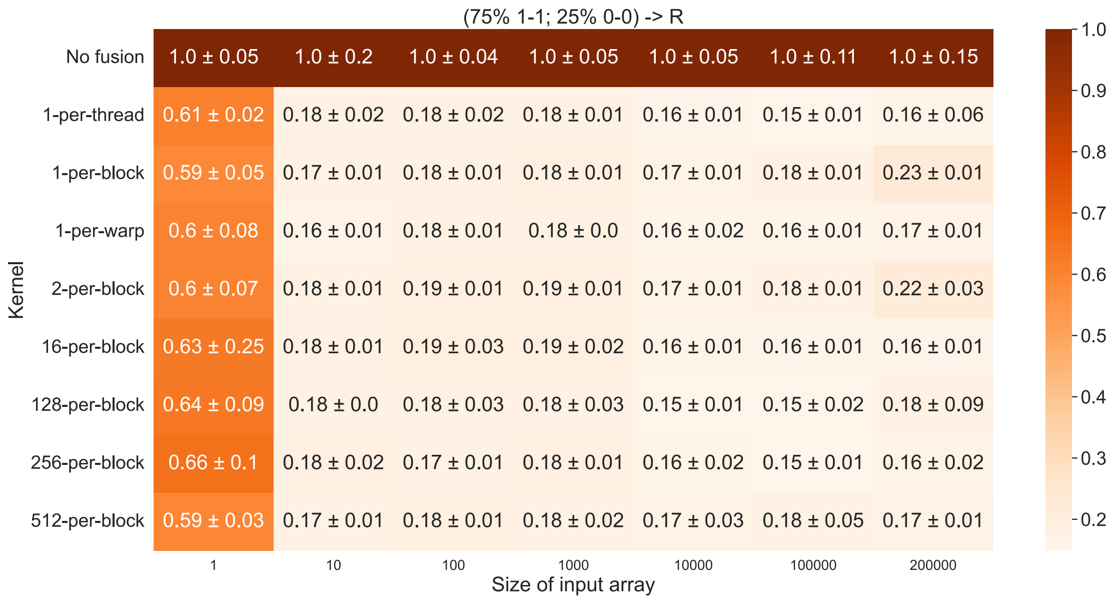
Kernel	1	10	100	1000	10000	100000	200000
No fusion	1.0 ± 0.12	1.0 ± 0.07	1.0 ± 0.07	1.0 ± 0.08	1.0 ± 0.06	1.0 ± 0.04	1.0 ± 0.03
1-per-thread	0.21 ± 0.03	0.21 ± 0.02	0.19 ± 0.02	0.16 ± 0.02	0.19 ± 0.01	0.59 ± 0.01	0.75 ± 0.01
1-per-block	0.2 ± 0.01	0.21 ± 0.02	0.19 ± 0.01	0.17 ± 0.01	0.2 ± 0.01	0.19 ± 0.01	0.17 ± 0.0
1-per-warp	0.2 ± 0.01	0.22 ± 0.01	0.19 ± 0.02	0.18 ± 0.02	0.19 ± 0.01	0.17 ± 0.01	0.15 ± 0.0
2-per-block	0.19 ± 0.01	0.22 ± 0.01	0.2 ± 0.02	0.19 ± 0.0	0.21 ± 0.01	0.18 ± 0.01	0.16 ± 0.03
16-per-block	0.19 ± 0.01	0.21 ± 0.01	0.2 ± 0.03	0.17 ± 0.0	0.19 ± 0.02	0.17 ± 0.01	0.16 ± 0.01
128-per-block	0.2 ± 0.01	0.21 ± 0.01	0.21 ± 0.01	0.2 ± 0.01	0.23 ± 0.04	0.18 ± 0.01	0.16 ± 0.03
256-per-block	0.19 ± 0.01	0.23 ± 0.06	0.21 ± 0.01	0.23 ± 0.04	0.27 ± 0.02	0.18 ± 0.01	0.17 ± 0.01
512-per-block	0.2 ± 0.02	0.22 ± 0.01	0.21 ± 0.02	0.24 ± 0.0	0.27 ± 0.01	0.18 ± 0.01	0.18 ± 0.01

Figure 6: Average run time of kernels on the ‘(100% 0-100) -> R’ benchmark, relative to the run time of the kernel without fusion.



Kernel	1	10	100	1000	10000
No fusion	1.0 ± 0.07	1.0 ± 0.06	1.0 ± 0.07	1.0 ± 0.14	1.0 ± 0.02
1-per-thread	0.25 ± 0.02	0.34 ± 0.03	0.44 ± 0.04	0.63 ± 0.05	0.24 ± 0.01
1-per-block	0.17 ± 0.02	0.21 ± 0.01	0.34 ± 0.02	0.56 ± 0.02	0.18 ± 0.03
1-per-warp	0.18 ± 0.01	0.21 ± 0.01	0.35 ± 0.02	0.37 ± 0.05	0.14 ± 0.04
2-per-block	0.16 ± 0.01	0.22 ± 0.01	0.39 ± 0.01	0.61 ± 0.02	0.18 ± 0.01
16-per-block	0.17 ± 0.01	0.23 ± 0.02	0.43 ± 0.04	0.68 ± 0.04	0.18 ± 0.01
128-per-block	0.18 ± 0.02	0.26 ± 0.03	0.79 ± 0.02	1.31 ± 0.05	0.31 ± 0.0
256-per-block	0.18 ± 0.01	0.26 ± 0.02	0.84 ± 0.05	2.48 ± 0.05	0.58 ± 0.0
512-per-block	0.18 ± 0.02	0.22 ± 0.02	0.53 ± 0.09	2.56 ± 0.03	0.6 ± 0.01

Figure 7: Average run time of kernels on the ‘(100% 1000-3000) -> R’ benchmark, relative to the run time of the kernel without fusion.



Kernel	1	10	100	1000	10000	100000	200000
No fusion	1.0 ± 0.05	1.0 ± 0.2	1.0 ± 0.04	1.0 ± 0.05	1.0 ± 0.05	1.0 ± 0.11	1.0 ± 0.15
1-per-thread	0.61 ± 0.02	0.18 ± 0.02	0.18 ± 0.02	0.18 ± 0.01	0.16 ± 0.01	0.15 ± 0.01	0.16 ± 0.06
1-per-block	0.59 ± 0.05	0.17 ± 0.01	0.18 ± 0.01	0.18 ± 0.01	0.17 ± 0.01	0.18 ± 0.01	0.23 ± 0.01
1-per-warp	0.6 ± 0.08	0.16 ± 0.01	0.18 ± 0.01	0.18 ± 0.0	0.16 ± 0.02	0.16 ± 0.01	0.17 ± 0.01
2-per-block	0.6 ± 0.07	0.18 ± 0.01	0.19 ± 0.01	0.19 ± 0.01	0.17 ± 0.01	0.18 ± 0.01	0.22 ± 0.03
16-per-block	0.63 ± 0.25	0.18 ± 0.01	0.19 ± 0.03	0.19 ± 0.02	0.16 ± 0.01	0.16 ± 0.01	0.16 ± 0.01
128-per-block	0.64 ± 0.09	0.18 ± 0.0	0.18 ± 0.03	0.18 ± 0.03	0.15 ± 0.01	0.15 ± 0.02	0.18 ± 0.09
256-per-block	0.66 ± 0.1	0.18 ± 0.02	0.17 ± 0.01	0.18 ± 0.01	0.16 ± 0.02	0.15 ± 0.01	0.16 ± 0.02
512-per-block	0.59 ± 0.03	0.17 ± 0.01	0.18 ± 0.01	0.18 ± 0.02	0.17 ± 0.03	0.18 ± 0.05	0.17 ± 0.01

Figure 8: Average run time of kernels on the '(75% 1-1; 25% 0-0) -> R' benchmark, relative to the run time of the kernel without fusion.

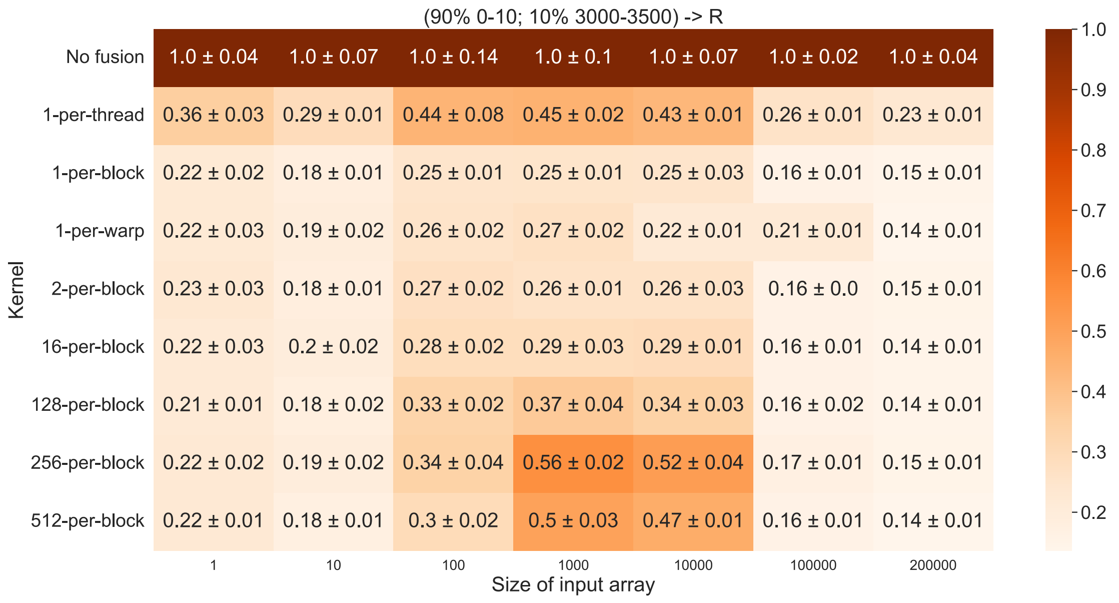
As with the previous benchmark, we see that there are kernels that have a better run time than the kernel without fusion. Two things stand out:

1. The last column contains very small values.
2. The bottom three kernels are slower than the situation without fusion for some inputs.

Both of these observations can be explained. The last column contains very low values, because the absolute run time of the kernel without fusion is very large. This indicates that there is a bottleneck in the kernel without fusion, that is not present in the kernels with fusion.

The reason that the some kernels are slower can be explained by looking at the expansion sizes again. The expected expansion size for an input element is 1500. This means that an ideal division over thread block threads would be $\lceil 1024/1500 \rceil = 1$. This is also visible in figure 7; the 1-per-block kernel has the best run time of all n-per-block kernels. The 128-per-block kernel has to perform $128 \cdot 1500 = 192,000$ expansions. As every thread does one expansion, $\lceil 192,000/1024 \rceil = 188$ passes are required per thread block per input.

(75% 1-1; 25% 0-0) $\mapsto$ R In this benchmark, the `expand` operation is used as a filter. Since this is one of the use-cases for `expand`, it is interesting to see whether there is an increase in performance in the fused kernel. The results are plotted in figure 8.



Kernel	1	10	100	1000	10000	100000	200000
No fusion	1.0 ± 0.04	1.0 ± 0.07	1.0 ± 0.14	1.0 ± 0.1	1.0 ± 0.07	1.0 ± 0.02	1.0 ± 0.04
1-per-thread	0.36 ± 0.03	0.29 ± 0.01	0.44 ± 0.08	0.45 ± 0.02	0.43 ± 0.01	0.26 ± 0.01	0.23 ± 0.01
1-per-block	0.22 ± 0.02	0.18 ± 0.01	0.25 ± 0.01	0.25 ± 0.01	0.25 ± 0.03	0.16 ± 0.01	0.15 ± 0.01
1-per-warp	0.22 ± 0.03	0.19 ± 0.02	0.26 ± 0.02	0.27 ± 0.02	0.22 ± 0.01	0.21 ± 0.01	0.14 ± 0.01
2-per-block	0.23 ± 0.03	0.18 ± 0.01	0.27 ± 0.02	0.26 ± 0.01	0.26 ± 0.03	0.16 ± 0.0	0.15 ± 0.01
16-per-block	0.22 ± 0.03	0.2 ± 0.02	0.28 ± 0.02	0.29 ± 0.03	0.29 ± 0.01	0.16 ± 0.01	0.14 ± 0.01
128-per-block	0.21 ± 0.01	0.18 ± 0.02	0.33 ± 0.02	0.37 ± 0.04	0.34 ± 0.03	0.16 ± 0.02	0.14 ± 0.01
256-per-block	0.22 ± 0.02	0.19 ± 0.02	0.34 ± 0.04	0.56 ± 0.02	0.52 ± 0.04	0.17 ± 0.01	0.15 ± 0.01
512-per-block	0.22 ± 0.01	0.18 ± 0.01	0.3 ± 0.02	0.5 ± 0.03	0.47 ± 0.01	0.16 ± 0.01	0.14 ± 0.01

Figure 9: Average run time of kernels on the '(90% 0-10; 10% 3000-3500) -> R' benchmark, relative to the run time of the kernel without fusion.

From the results it is clear that all kernels have a run time that is about five times better than the situation without fusion. An interesting observation for this benchmark is that the 1-per-thread kernel performs very well.This makes sense, because every input element does at most one expansion. Moreover, most n-per-block kernel do not make efficient use of the available threads, because some threads will be idle if n is less than 1024.

(90% 0-10; 10% 3000-3500) $\mapsto$ R In this benchmark, an element has a 90% chance to expand to a small size, and a 10% chance to expand to a large size. All elements are permuted to a random target index. This probabilistic expansion size makes it so that some thread blocks have much more work to do than others. Therefore, it is expected that a kernel that takes on multiple elements of the input (i.e. a n-per-block kernel) performs better on this benchmark. The results are plotted in figure 9.

The first observation we can make from the results is that, again, all kernels offer a speedup compared to the situation without fusion. The next observation we can make is that the 1-per-thread kernel is the slowest of the kernels. This is likely because this kernel is not able to handle the irregular expansion pattern present in this benchmark. As one in ten elements will expand to a large size, a tenth of all threads will have a lot of work. All smaller threads then need to wait for the thread with the large expansion size to finish. Since smaller expansions take much less time, 90% of threads will be waiting about two-thirds of the time. This especially becomes a bottleneck on bigger input sizes, on which the kernel

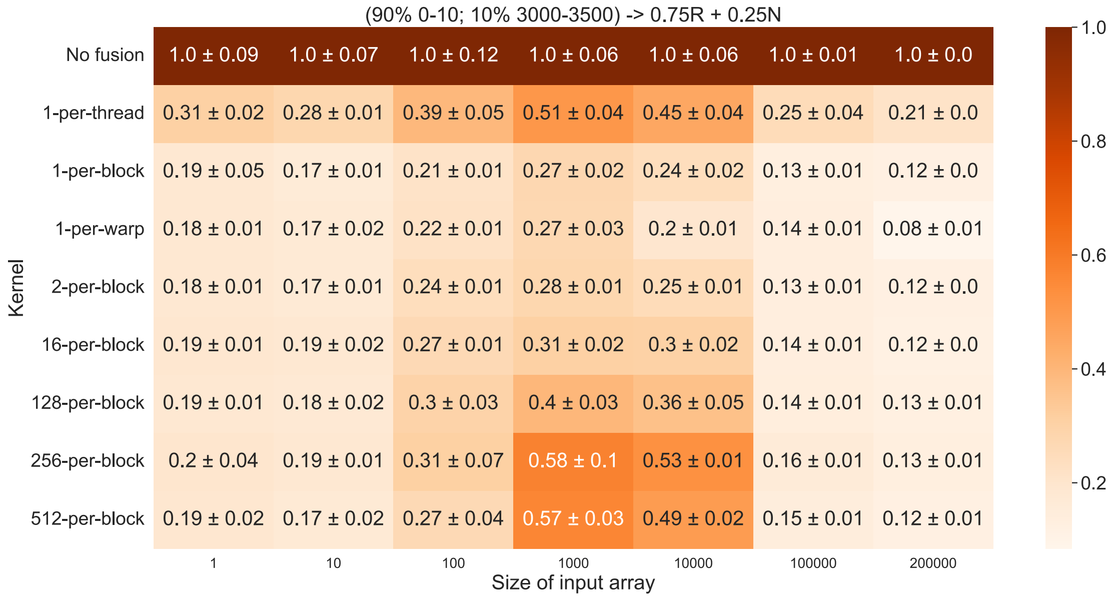


Figure 10: Average run time of kernels on the '(90% 0-10; 10% 3000-3500) -> 0.75R + 0.25N' benchmark, relative to the run time of the kernel without fusion.

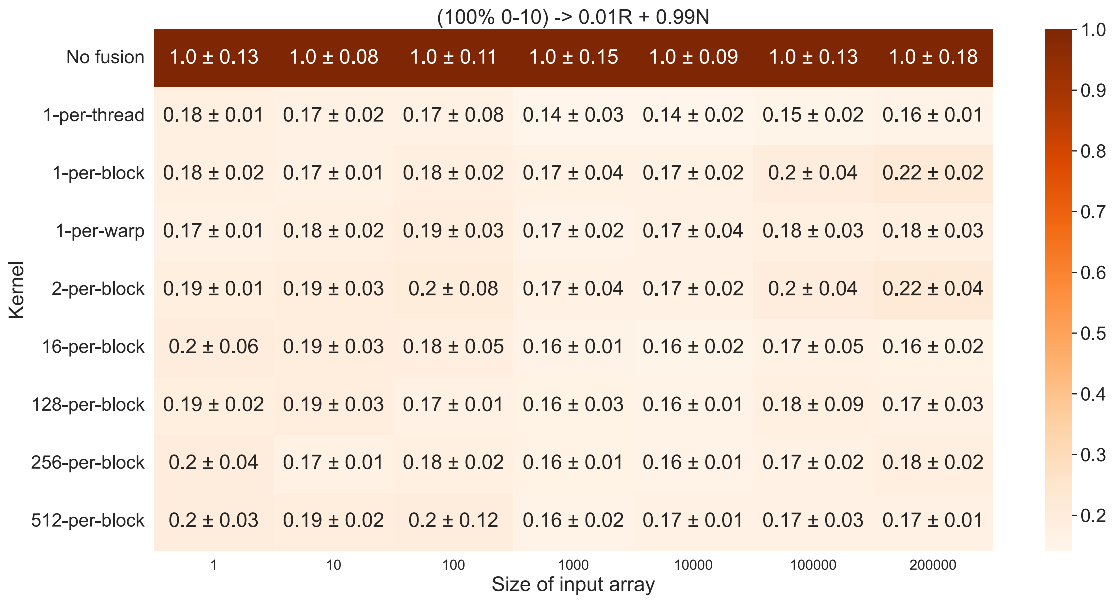
needs to do multiple passes to fully process all input elements.

Finally, we see that the 128-, 256- and 512-per-block have a larger run time than the other n-per-block kernels. The reason for this is likely analogous to that of the 1-per-thread kernel. Focussing on the 128-per-block kernel, there will be about 12 input elements expanding to a large size in every thread block. This means that every thread block has an expected amount of total expansions of $12 \cdot 3250 + 116 \cdot 50 = 44800$, divided over 1024 threads. All threads thus need to process at least $\lfloor 44800/1024 \rfloor = 43$ expansions. This amount of passes could slow down the kernel.

(90% 0-10; 10% 3000-3500) $\mapsto$ 0.75$\cdot$ R + 0.25$\cdot$ N In this benchmark, an element has an 90% chance to expand to a small size, and a 10% chance to expand to a large size. Moreover, some elements are permuted to an index of `Nothing`, thus filtered out. This is to evaluate performance when the `permute` operation is also used to filter. Note that the `expand` operation is also used as a filter, as the expansion size can become zero. The results are plotted in figure 10.

In this figure, we see that the results are quite similar to those in 9. This leads us to believe that the performance for this benchmark is mostly influenced by the expansions.

(100% 0-10) $\mapsto$ 0.01$\cdot$ R + 0.99$\cdot$ N This benchmark is created specifically to analyse the performance of the expanding part of the kernel. Because almost all elements are permuted to `Nothing`, the writes performed by a permutation should



Kernel	1	10	100	1000	10000	100000	200000
No fusion	1.0 ± 0.13	1.0 ± 0.08	1.0 ± 0.11	1.0 ± 0.15	1.0 ± 0.09	1.0 ± 0.13	1.0 ± 0.18
1-per-thread	0.18 ± 0.01	0.17 ± 0.02	0.17 ± 0.08	0.14 ± 0.03	0.14 ± 0.02	0.15 ± 0.02	0.16 ± 0.01
1-per-block	0.18 ± 0.02	0.17 ± 0.01	0.18 ± 0.02	0.17 ± 0.04	0.17 ± 0.02	0.2 ± 0.04	0.22 ± 0.02
1-per-warp	0.17 ± 0.01	0.18 ± 0.02	0.19 ± 0.03	0.17 ± 0.02	0.17 ± 0.04	0.18 ± 0.03	0.18 ± 0.03
2-per-block	0.19 ± 0.01	0.19 ± 0.03	0.2 ± 0.08	0.17 ± 0.04	0.17 ± 0.02	0.2 ± 0.04	0.22 ± 0.04
16-per-block	0.2 ± 0.06	0.19 ± 0.03	0.18 ± 0.05	0.16 ± 0.01	0.16 ± 0.02	0.17 ± 0.05	0.16 ± 0.02
128-per-block	0.19 ± 0.02	0.19 ± 0.03	0.17 ± 0.01	0.16 ± 0.03	0.16 ± 0.01	0.18 ± 0.09	0.17 ± 0.03
256-per-block	0.2 ± 0.04	0.17 ± 0.01	0.18 ± 0.02	0.16 ± 0.01	0.16 ± 0.01	0.17 ± 0.02	0.18 ± 0.02
512-per-block	0.2 ± 0.03	0.19 ± 0.02	0.2 ± 0.12	0.16 ± 0.02	0.17 ± 0.01	0.17 ± 0.03	0.17 ± 0.01

Figure 11: Average run time of kernels on the '(100% 0-10) -> 0.01R + 0.99N' benchmark, relative to the run time of the kernel without fusion.

not lead to any overhead. Moreover, because the expected expansion size is 5, all expansion computations should mostly fit within one thread block when the n-per-block kernels. The results of this benchmark are plotted in figure 11. We see that these results are quite similar to those in figure 5. This indicates that the permutation has a small effect on the total run time of the kernel.

5.2 Realistic Benchmarks

In the following paragraphs, we discuss some realistic benchmarks that perform a permutation after an expansion. These benchmarks should give an idea of the performance increase that the fusion is able to achieve in a real algorithm.

Sieve of Eratosthenes The first benchmark is the Sieve of Eratosthenes, which is also used in the paper "Data-parallel flattening by expansion" [6]. The algorithm calculates the primes smaller than some given n. In listing 10 of appendix A, an adaptation of the Accelerate implementation that can be found in the Accelerate documentation is given. This adaptation uses the new `permute'` function. The results of this benchmark can be found in figure 12.

Inspecting the results of this benchmark, we can observe the following. For a limit of at most 100,000, all kernels are faster than the situation without fusion. However, for a limit of 1,000,000 and up, only the 1-per-block kernel is faster than the situation without fusion. This is likely because the input of the expand is heavily unbalanced; the expansion size of later elements is many times smaller

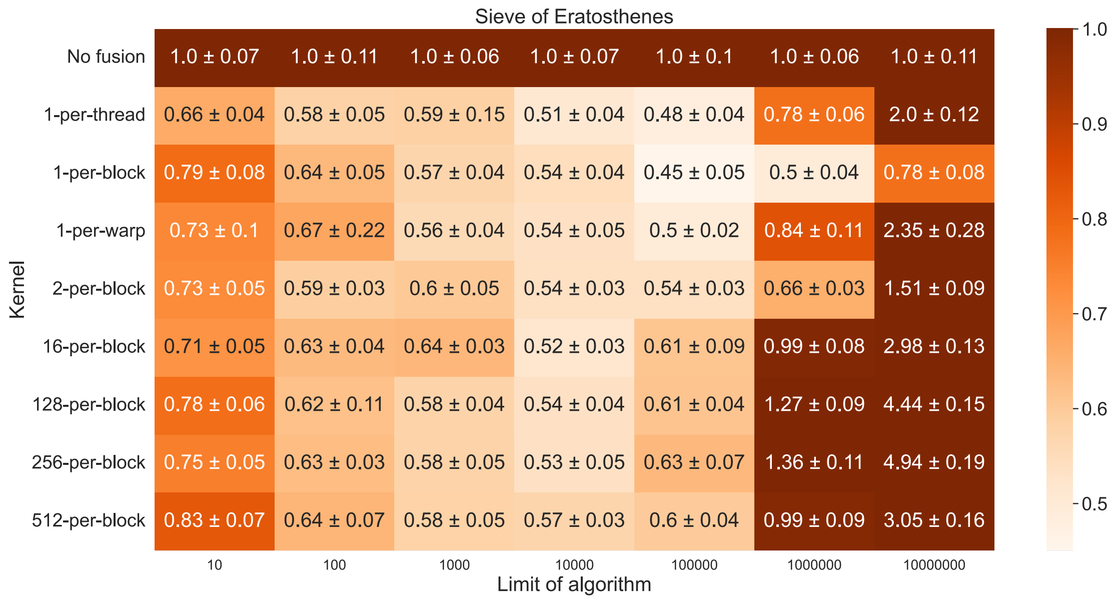


Figure 12: Average run time of kernels on the 'Sieve of Eratosthenes' benchmark, relative to the run time of the kernel without fusion.

than the expansion size of earlier elements. This can be easily calculated when taking a concrete example.

Consider the 2-per-block kernel, with a limit of 10,000,000. As we see in listing 10, the size function is `size p = (c2 - p) ‘quot‘ p`. The array of source values for the `expand` operation will contain all primes up to `c1`. We can compute the sizes for the last iteration, in which case `c2` will equal `n + 1`. Some of these sizes are as follows:

$$
\begin{aligned}
\texttt{size 2} &= & (10,000,001 - 2)/2 && \approx 5,000,000 \\
\texttt{size 3} &= & (10,000,001 - 3)/3 && \approx 3,333,333 \\
\texttt{size 5} &= & (10,000,001 - 5)/5 && \approx 2,000,000 \\
\texttt{size 7} &= & (10,000,001 - 7)/7 && \approx 1,428,570 \\
\texttt{size 11} &= & (10,000,001 - 11)/11 && \approx 909,090 \\
\texttt{size 13} &= & (10,000,001 - 13)/13 && \approx 769,229
\end{aligned}
$$

We know that the primes 2 and 3 will be expanded by the first thread block, the primes 5 and 7 will be expanded by the second block and that the primes 11 and 13 will be expanded by the third block. This means that the total number of expansions s_i for block i will be:

$$
\begin{aligned}
s_0 &= 5,000,000 + 3,333,333 && = 8,333,333 \\
s_1 &= 2,000,000 + 1,428,570 && = 3,428,570 \\
s_2 &= 909,090 + 769,229 && = 1,678,319
\end{aligned}
$$

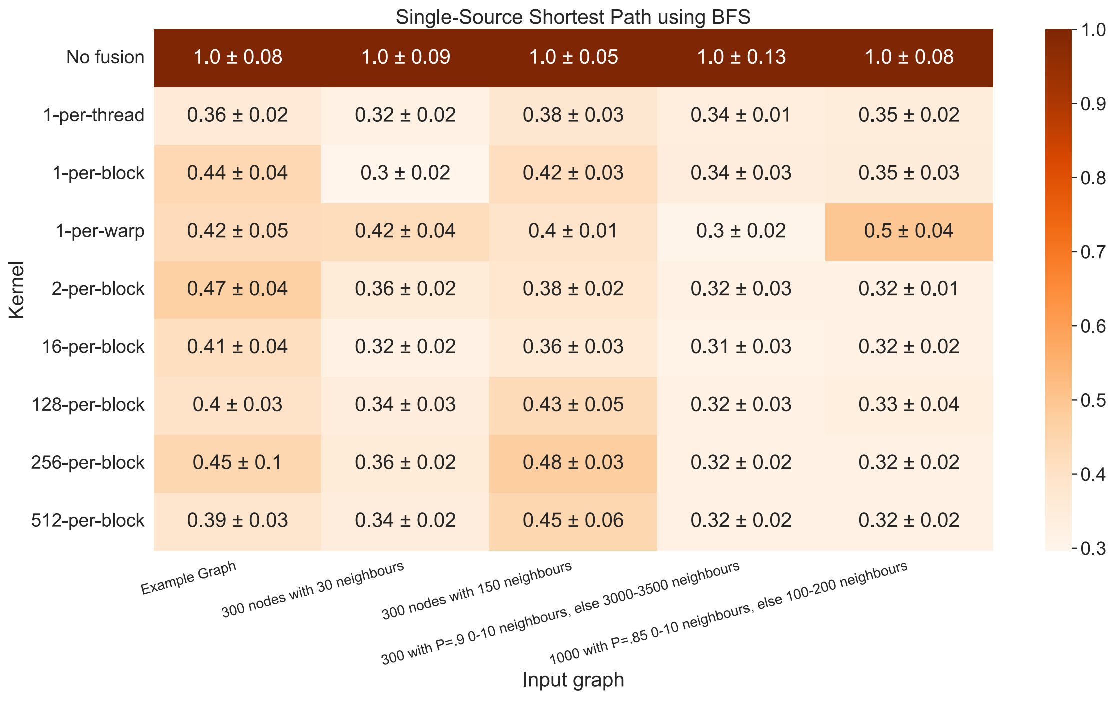


Figure 13: Average run time of kernels on the 'Single-Source Shortest Path using BFS' benchmark, relative to the run time of the kernel without fusion.

This shows that the first thread block needs to do 2.4 times as much work as the second block, and 5 times as much work as the third thread block. We thus see that some thread blocks will form a big bottleneck, because the input is pathologically unbalanced. The 1-per-block kernel however, does the most expansions in its first thread block, namely 5,000,000. While this also cannot be processed in one pass using only 1,024 threads, the first block should be able to process its elements about 40% faster, which is in line with the benchmark results.

BFS in a Graph The second benchmark is a graph algorithm. It is an implementation of a single-source multiple-destination shortest path algorithm, using breadth-first search. In this algorithm, the `expand`-operation is used to find neighbours of explored nodes. We can then use the `permute'`-operation to update the parent value of these neighbouring nodes, thereby marking them as explored. The loop runs until the vector of parents is no longer updated. The code can be found in listing 11 of appendix A. From the results of in figure 13, we see that for various graph structures, the fused kernel is faster than the unfused kernel.

Rendering 3D Models The third benchmark is an algorithm for rendering in 3D. This use case is mentioned in the paper introducing expand [6], but not implemented as the Futhark implementation of `scatter` does not have a combination

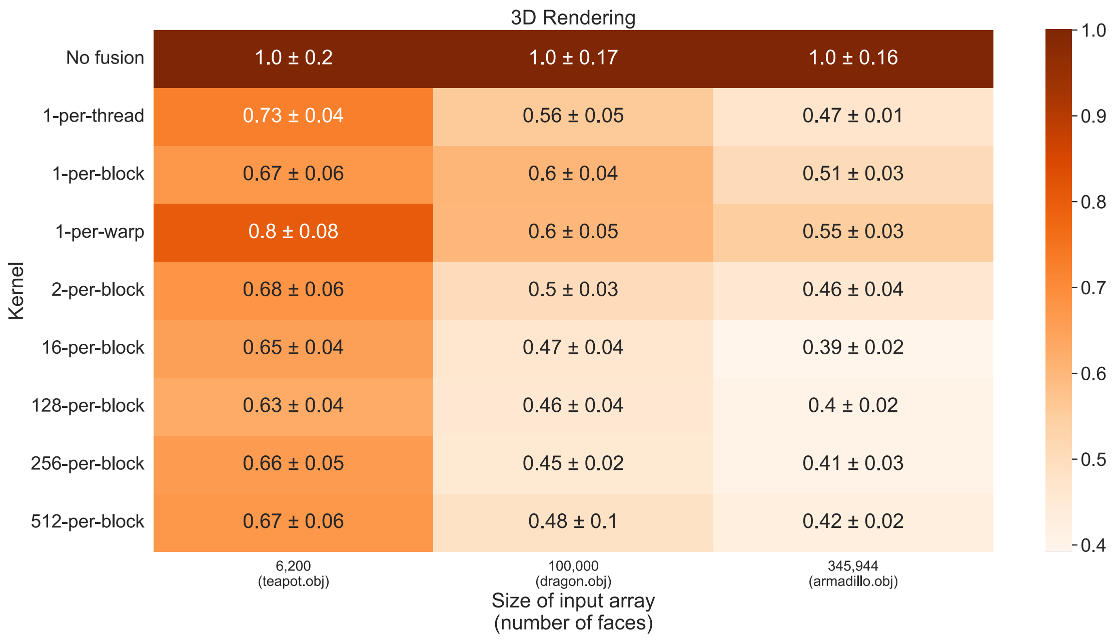


Figure 14: Average run time of kernels on the '3D Rendering' benchmark, relative to the run time of the kernel without fusion.

function. As the Accelerate implementation of `permute'` does provide this, we can use the following process for rendering 3D objects.

The benchmark first projects a set of 3D triangle corner coordinates onto a screen, resulting in a set of 2D triangle corner coordinates. These coordinates now form the 2D triangles that should be drawn on the canvas. The projection is implemented using the theory from *Fundamentals of Computer Graphics (4. ed.)* [40]. Using the `expand` operation, we can find the endpoints of the horizontal lines that make up the triangles. Using yet another `expand` operation, the endpoints of these lines can be translated to all points on the horizontal lines. This results in all 2D points that make up all 2D triangles. We then use `permute'` to plot these points on the canvas, where the point with the lowest z-index is drawn on top. An example of such a rendering can be found in figure 15.

As a benchmark, we have rendered multiple models. We have used the Utah teapot, consisting of 6,200 faces. We have also used the Stanford dragon, consisting of 100,000 faces. Furthermore, the Stanford armadillo consisting of 345,944 faces was used. These models are well-known 3D test models. The results of this benchmark are plotted in figure 14. In this figure, we can see that the run time improves when using the fused kernel. The reason that the increase is so much bigger for the armadillo model, is because the absolute time that it takes the unfused version is much greater. All other absolute kernel times stay in the same range as for the other inputs.

Moreover, we see that it is beneficial to use to use a kernel that expands multiple

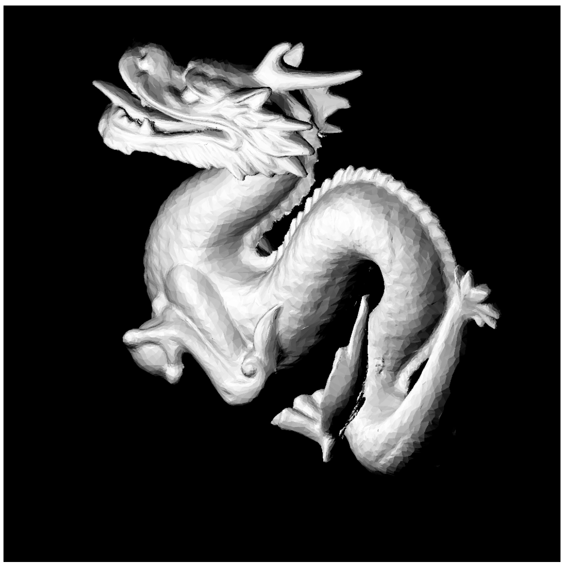
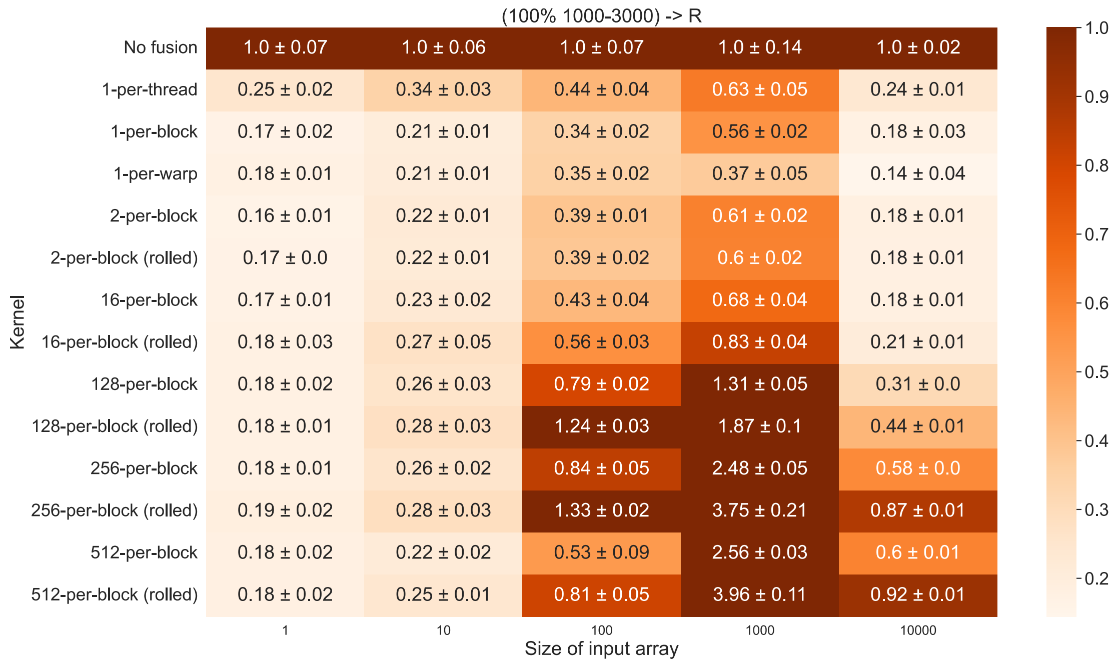


Figure 15: Stanford dragon, rendered using the `expand` and `permute` operations. The dragon is white, and shaded with Lambertian shading.

input elements using one thread block. This indicates that most input elements expand to a small number of target elements. This makes sense, as the canvas size is 1080 by 1080. Most triangles take up only a small portion of this canvas, otherwise the model would not fully fit on the visible part of it. Because triangles expand to a small number of pixels, it is beneficial to handle multiple triangles in one thread block.

Note that models with more faces will result in smaller expansions. This is because more triangles are rendered on a canvas of the same size. To fully show the model, all triangles must thus be drawn smaller. Given that a pixel has a static size, every triangle spans less pixels. In the case of a larger model, the expansion size will thus be smaller, however the number of source elements will be higher.



Kernel	1	10	100	1000	10000
No fusion	1.0 ± 0.07	1.0 ± 0.06	1.0 ± 0.07	1.0 ± 0.14	1.0 ± 0.02
1-per-thread	0.25 ± 0.02	0.34 ± 0.03	0.44 ± 0.04	0.63 ± 0.05	0.24 ± 0.01
1-per-block	0.17 ± 0.02	0.21 ± 0.01	0.34 ± 0.02	0.56 ± 0.02	0.18 ± 0.03
1-per-warp	0.18 ± 0.01	0.21 ± 0.01	0.35 ± 0.02	0.37 ± 0.05	0.14 ± 0.04
2-per-block	0.16 ± 0.01	0.22 ± 0.01	0.39 ± 0.01	0.61 ± 0.02	0.18 ± 0.01
2-per-block (rolled)	0.17 ± 0.0	0.22 ± 0.01	0.39 ± 0.02	0.6 ± 0.02	0.18 ± 0.01
16-per-block	0.17 ± 0.01	0.23 ± 0.02	0.43 ± 0.04	0.68 ± 0.04	0.18 ± 0.01
16-per-block (rolled)	0.18 ± 0.03	0.27 ± 0.05	0.56 ± 0.03	0.83 ± 0.04	0.21 ± 0.01
128-per-block	0.18 ± 0.02	0.26 ± 0.03	0.79 ± 0.02	1.31 ± 0.05	0.31 ± 0.0
128-per-block (rolled)	0.18 ± 0.01	0.28 ± 0.03	1.24 ± 0.03	1.87 ± 0.1	0.44 ± 0.01
256-per-block	0.18 ± 0.01	0.26 ± 0.02	0.84 ± 0.05	2.48 ± 0.05	0.58 ± 0.0
256-per-block (rolled)	0.19 ± 0.02	0.28 ± 0.03	1.33 ± 0.02	3.75 ± 0.21	0.87 ± 0.01
512-per-block	0.18 ± 0.02	0.22 ± 0.02	0.53 ± 0.09	2.56 ± 0.03	0.6 ± 0.01
512-per-block (rolled)	0.18 ± 0.02	0.25 ± 0.01	0.81 ± 0.05	3.96 ± 0.11	0.92 ± 0.01

Figure 16: Average run time of kernels on the '(100% 1000-3000) -> R' benchmark, relative to the run time of the kernel without fusion. The results contain some overlap with figure 7.

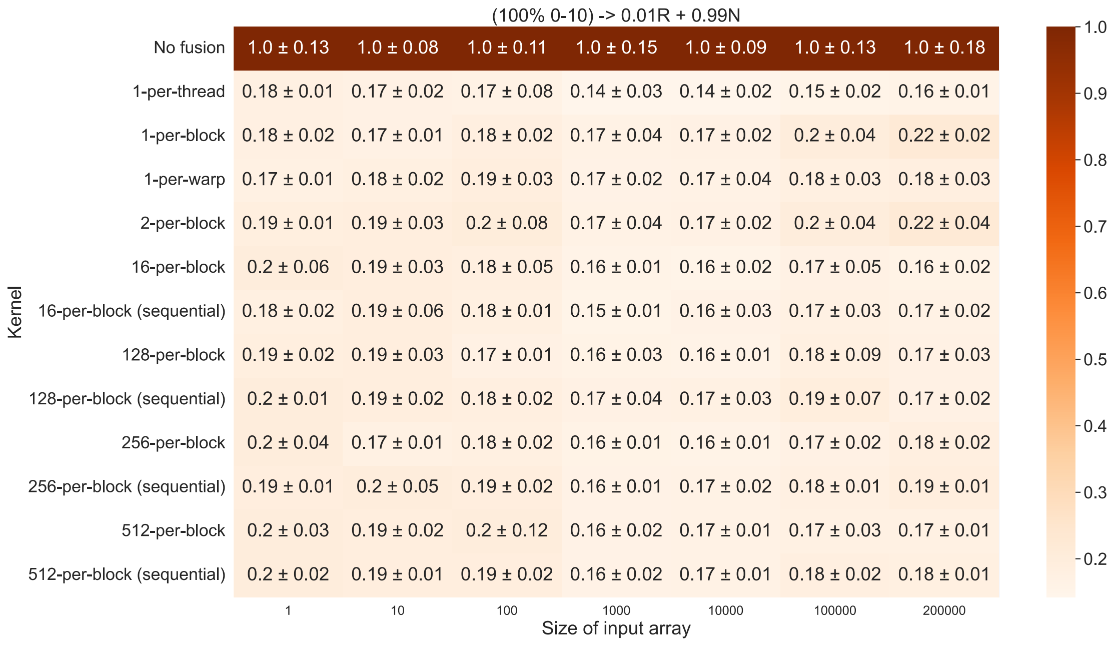
5.3 Unrolling Binary Search

To test the effectiveness of unrolling the binary search loop, we have performed some benchmarks using the kernel that unrolls the binary search and also using the kernel that uses a loop. From the results of all benchmarks, we can see that the unrolled version always performs equally as well as the rolled version of binary search. In some benchmarks, the performance of the unrolled version is better.

One benchmark in which this is visible is in the benchmark that expands to big sizes, that is (100% 1000-3000) $\mapsto$ R. In figure 16, the same results as in figure 7 are plotted, but the run times for the kernel with a rolled binary search are also shown. In this table we see that the unrolled version is always faster than the rolled version.

5.4 Parallel Prefix Sum

To test the effectiveness of using a parallel version of the prefix sum, we have performed the benchmarks using a kernel that uses a sequential sum and using a kernel that uses a parallel version. It is not tested for the 2-per-block kernel, given that it does not make sense to parallelise the sum of two elements. From the results in figure 17, we see that the difference in performance seems negligible in most cases. In cases with a reasonable amount of input elements and where there



(100% 0-10) -> 0.01R + 0.99N

Kernel	1	10	100	1000	10000	100000	200000
No fusion	1.0 ± 0.13	1.0 ± 0.08	1.0 ± 0.11	1.0 ± 0.15	1.0 ± 0.09	1.0 ± 0.13	1.0 ± 0.18
1-per-thread	0.18 ± 0.01	0.17 ± 0.02	0.17 ± 0.08	0.14 ± 0.03	0.14 ± 0.02	0.15 ± 0.02	0.16 ± 0.01
1-per-block	0.18 ± 0.02	0.17 ± 0.01	0.18 ± 0.02	0.17 ± 0.04	0.17 ± 0.02	0.2 ± 0.04	0.22 ± 0.02
1-per-warp	0.17 ± 0.01	0.18 ± 0.02	0.19 ± 0.03	0.17 ± 0.02	0.17 ± 0.04	0.18 ± 0.03	0.18 ± 0.03
2-per-block	0.19 ± 0.01	0.19 ± 0.03	0.2 ± 0.08	0.17 ± 0.04	0.17 ± 0.02	0.2 ± 0.04	0.22 ± 0.04
16-per-block	0.2 ± 0.06	0.19 ± 0.03	0.18 ± 0.05	0.16 ± 0.01	0.16 ± 0.02	0.17 ± 0.05	0.16 ± 0.02
16-per-block (sequential)	0.18 ± 0.02	0.19 ± 0.06	0.18 ± 0.01	0.15 ± 0.01	0.16 ± 0.03	0.17 ± 0.03	0.17 ± 0.02
128-per-block	0.19 ± 0.02	0.19 ± 0.03	0.17 ± 0.01	0.16 ± 0.03	0.16 ± 0.01	0.18 ± 0.09	0.17 ± 0.03
128-per-block (sequential)	0.2 ± 0.01	0.19 ± 0.02	0.18 ± 0.02	0.17 ± 0.04	0.17 ± 0.03	0.19 ± 0.07	0.17 ± 0.02
256-per-block	0.2 ± 0.04	0.17 ± 0.01	0.18 ± 0.02	0.16 ± 0.01	0.16 ± 0.01	0.17 ± 0.02	0.18 ± 0.02
256-per-block (sequential)	0.19 ± 0.01	0.2 ± 0.05	0.19 ± 0.02	0.16 ± 0.01	0.17 ± 0.02	0.18 ± 0.01	0.19 ± 0.01
512-per-block	0.2 ± 0.03	0.19 ± 0.02	0.2 ± 0.12	0.16 ± 0.02	0.17 ± 0.01	0.17 ± 0.03	0.17 ± 0.01
512-per-block (sequential)	0.2 ± 0.02	0.19 ± 0.01	0.19 ± 0.02	0.16 ± 0.02	0.17 ± 0.01	0.18 ± 0.02	0.18 ± 0.01

Figure 17: Average run time of kernels on the '(100% 0-10) -> 0.01R + 0.99N' benchmark, relative to the run time of the kernel without fusion. The results contain some overlap with figure 11.

is a small amount of variance, and a distinction can thus be made, we see that when a parallel version of the prefix sum is used the performance is increased by a small amount.

6 Conclusions

This thesis has brought several contributions to the field of data-parallel array languages. The first contribution, the fusion of `expand` with `permute` in an AST, merely builds on previous work, yet provides a fundamental starting point for the code generation of the fusion.

The second contribution is the generation of low-level code that performs a permutation after an expansion without producing an intermediate result. We have discussed several possible implementations for such low-level code, as well as their advantages and drawbacks. The discussion should be applicable to generic parallel array languages that compile to a different low-level language. An example implementation of this low-level code generation is given in the Accelerate language.

Finally, benchmarks have been run on the implementation in Accelerate. Using the results from these benchmarks, it is possible to draw conclusions about the effectiveness of the fused kernel. As can be seen in the benchmark results, using a fused kernel is always better than using two operations. Moreover, in most cases the performance was twice as good. In many cases the performance was even up to five times as good as the situation without fusion. Especially the 1-per-block kernel seems to be an improvement over the version without fusion.

The reason that the n-per-block kernels become slower on big inputs, is likely because the amount of work they need to do depends heavily on the chosen n. When the n increases, the increased amount of work still is done by only 1024 threads. This is a problem if the total amount of work exceeds this number.

6.1 Future Work

There are a number of possibilities for future work. One option is to randomise the order in which input elements are assigned to blocks in the n-per-block kernels. This might reduce the effect that pathological inputs like the Sieve of Eratosthenes have on the total run time. This randomisation should not influence the outcome of a program, given that that every expanded element will be permuted and that the permutation index is generated during the expansion.

Another option is to see whether performance is increased when every thread performs one expansion. To divide work like this, a prefix sum over all input elements will be necessary, after which all expansions of all input elements can be divided over all available threads. This could lead to a 'perfect' balancing of workload, but does require an intensive scan that uses global memory. It will also require an extra pass over the input, to compute the expansion sizes of all input elements.

Appendix A Benchmark Implementations

A.1 Sieve of Eratosthenes

```haskell
primes :: Exp Int -> Acc (Vector Int)
primes n = afst loop
  where
    c0    = unit 2
    a0    = use $ fromList (Z:.0) []
    limit = truncate $ sqrt $ fromIntegral (n+1)
    loop  = awhile
              (\(T2 _ c) -> map (< n+1) c)
              (\(T2 old c) ->
                let c1 = the c
                    c2 = c1 < limit ? ( c1*c1, n+1 )
                    --
                    sieves =
                      let size p = (c2 - p) `quot` p
                          get p i = (2+i)*p
                          put x = x >= 0 && x < m
                            ? (Just_ (I1 x), Nothing_))
                      in
                      map (\x -> T2 (put x) 0)
                        $ map (subtract c1)
                        $ expand size get old
                    --
                    new =
                      let m     = c2-c1
                      in
                      afst
                        $ filter (> 0)
                        $ permute' const
                            (enumFromN (I1 m) c1)
                        $ sieves
                in
                T2 (old ++ new) (unit c2))
              (T2 a0 c0)
```

Listing 10: Sieve of Erastosthenes.

A.2 BFS in a Graph

```haskell
bfs :: Exp Int -> Exp Int -> (Exp Int -> Exp Int)
    -> (Exp Int -> Exp Int -> Exp Int)
    -> Acc (Vector (Maybe Int))
bfs src totalNodes numNeighbours getNeighbour = afst loop
  where
    parentsI =
        generate (I1 totalNodes)
        (\(I1 i) -> cond (i == src) (Just_ src) Nothing_)
    parentsN = fill (I1 totalNodes) Nothing_
    loop =
      awhile
        (\(T2 parents oldParents) ->
          map not $ and $ zipWith (==) parents oldParents)
        (\(T2 parents oldParents) ->
          let
              known = afst $ compact
                (map (not . (match isNothing)) parents)
                (enumFromN (I1 totalNodes) 0)
              e = expand (match numNeighbours)
                  (\elem i -> T2 (Just_ $ I1 $
                    (match getNeighbour) elem i)
                  (Just_ elem)) known
              (neighbours, newParents) = unzip e
              perm (I1 i) = neighbours !! i
              p = permute (match (<|>)) parents perm
                  newParents
          in T2 p parents
        )
        (T2 parentsI parentsN)
```
Listing 11: Single-source multiple-destination shortest path algorithm using BFS.

References

[1] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, M. Carro and J. H. Reppy, Eds., ACM, 2011, pp. 3–14. DOI: `10.1145/1926354.1926358`. [Online]. Available: `https://doi.org/10.1145/1926354.1926358`.

[2] The Futhark Hackers, *Futhark*. [Online]. Available: `https://github.com/diku-dk/futhark`.

[3] G. Mainland and G. Morrisett, "Nikola: Embedding compiled gpu functions in haskell," *SIGPLAN Not.*, vol. 45, no. 11, pp. 67–78, Sep. 2010, ISSN: 0362-1340. DOI: `10.1145/2088456.1863533`. [Online]. Available: `https://doi.org/10.1145/2088456.1863533`.

[4] J. Svensson, M. Sheeran, and K. Claessen, "Obsidian: A domain specific embedded language for parallel programming of graphics processors," in *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, ser. IFL'08, Hatfield, UK: Springer-Verlag, 2008, pp. 156–173, ISBN: 9783642244513.

[5] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional GPU programs," in *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, G. Morrisett and T. Uustalu, Eds., ACM, 2013, pp. 49–60. DOI: `10.1145/2500365.2500595`. [Online]. Available: `https://doi.org/10.1145/2500365.2500595`.

[6] M. Elsman, T. Henriksen, and N. G. W. Serup, "Data-parallel flattening by expansion," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, J. Gibbons, Ed., ACM, 2019, pp. 14–24. DOI: `10.1145/3315454.3329955`. [Online]. Available: `https://doi.org/10.1145/3315454.3329955`.

[7] S. Marlow *et al.*, "Haskell 2010 language report," Nov. 2010. [Online]. Available: `http://www.haskell.org/definition/haskell2010.pdf`.

[8] T. L. McDonell, "Optimising purely functional gpu programs," Ph.D. dissertation, University of New South Wales, 2015.

[9] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. L. P. Jones, "Vectorisation avoidance," in *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, J. Voigtländer, Ed., ACM, 2012, pp. 37–48. DOI: `10.1145/2364506.2364512`. [Online]. Available: `https://doi.org/10.1145/2364506.2364512`.

[10] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw, "Data-only flattening for nested data parallelism," *SIGPLAN Not.*, vol. 48, no. 8, pp. 81–92, Feb. 2013, ISSN: 0362-1340. DOI: `10.1145/2517327.2442525`. [Online]. Available: `https://doi.org/10.1145/2517327.2442525`.

[11] P. Hudak, "Building domain-specific embedded languages," *Acm computing surveys (csur)*, vol. 28, no. 4es, 196–es, 1996.

[12] J. Gibbons and N. Wu, "Folding domain-specific languages: Deep and shallow embeddings (functional pearl)," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, J. Jeuring and M. M. T. Chakravarty, Eds., ACM, 2014, pp. 339–347. DOI: `10.1145/2628136.2628138`. [Online]. Available: `https://doi.org/10.1145/2628136.2628138`.

[13] A. Azurat and I. Prasetya, "A survey on embedding programming logics in a theorem prover," Mar. 2002.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 Classes*, ser. SIGGRAPH '08, Los Angeles, California: Association for Computing Machinery, 2008, ISBN: 9781450378451. DOI: `10.1145/1401132.1401152`. [Online]. Available: `https://doi.org/10.1145/1401132.1401152`.

[15] NVIDIA Corporation, "CUDA C++ Programming Guide," Whitepaper, Feb. 2023. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[16] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engg.*, vol. 12, no. 3, pp. 66–73, May 2010, ISSN: 1521-9615.

[17] K. Claessen, M. Sheeran, and B. J. Svensson, "Expressive array constructs in an embedded gpu kernel programming language," in *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, ser. DAMP '12, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 21–30, ISBN: 9781450311175. DOI: `10.1145/2103736.2103740`. [Online]. Available: `https://doi.org/10.1145/2103736.2103740`.

[18] M. Harris, *Cuda pro tip: Do the kepler shuffle*, Aug. 2022. [Online]. Available: `https://developer.nvidia.com/blog/cuda-pro-tip-kepler-shuffle/`.

[19] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages," in *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, ser. LFP '82, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1982, pp. 1–10, ISBN: 0897910826. DOI: `10.1145/800068.802129`. [Online]. Available: `https://doi.org/10.1145/800068.802129`.

[20] T. A. Proebsting, "Optimizing an ansi c interpreter with superoperators," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 322–332, ISBN: 0897916921. DOI: `10.1145/199448.199526`. [Online]. Available: `https://doi.org/10.1145/199448.199526`.

[21] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986, ISSN: 0001-0782. DOI: `10.1145/7902.7904`. [Online]. Available: `https://doi.org/10.1145/7902.7904`.

[22] D. B. Loveman, "Program improvement by source-to-source transformation," *J. ACM*, vol. 24, no. 1, pp. 121–145, Jan. 1977, ISSN: 0004-5411. DOI: `10.1145/321992.322000`. [Online]. Available: `https://doi.org/10.1145/321992.322000`.

[23] P. Wadler, "Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 45–52, ISBN: 0897911423. DOI: `10.1145/800055.802020`. [Online]. Available: `https://doi.org/10.1145/800055.802020`.

[24] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93, Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 223–232, ISBN: 089791595X. DOI: `10.1145/165180.165214`. [Online]. Available: `https://doi.org/10.1145/165180.165214`.

[25] M. M. T. Chakravarty and G. Keller, "Functional array fusion," in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '01, Florence, Italy: Association for Computing Machinery, 2001, pp. 205–216, ISBN: 1581134150. DOI: `10.1145/507635.507661`. [Online]. Available: `https://doi.org/10.1145/507635.507661`.

[26] M. Boehm, B. Reinwald, D. Hutchison, A. V. Evfimievski, and P. Sen, *On optimizing operator fusion plans for large-scale machine learning in systemml*, 2018. arXiv: `1801.00829 [cs.DB]`.

[27] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation.* Cambridge, MA, USA: MIT Press, 1991, ISBN: 0262530864.

[28] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '11, Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 3–14,

ISBN: 9781450304863. DOI: `10.1145/1926354.1926358`. [Online]. Available: `https://doi.org/10.1145/1926354.1926358`.

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar. 2004, pp. 75–88.

[30] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton, "Type-safe runtime code generation: Accelerate to LLVM," in *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, B. Lippmeier, Ed., ACM, 2015, pp. 201–212. DOI: `10.1145/2804302.2804313`. [Online]. Available: `https://doi.org/10.1145/2804302.2804313`.

[31] J. Cheney and R. Hinze, "First-class phantom types," 2003.

[32] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996, ISSN: 0001-0782. DOI: `10.1145/227234.227246`. [Online]. Available: `https://doi.org/10.1145/227234.227246`.

[33] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986, ISSN: 0001-0782. DOI: `10.1145/7902.7903`. [Online]. Available: `https://doi.org/10.1145/7902.7903`.

[34] G. E. Blelloch, "Nesl: A nested data-parallel language (version 3.1)," USA, Tech. Rep., 1995.

[35] G. E. Blelloch, *Vector Models for Data-Parallel Computing.* Cambridge, MA, USA: MIT Press, 1990, ISBN: 026202313X.

[36] J. Luitjens, *Faster parallel reductions on kepler*, Aug. 2022. [Online]. Available: `https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/`.

[37] NVIDIA Corporation and J. Wang, "NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110/210," Whitepaper, 2014, p. 11. [Online]. Available: `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`.

[38] NVIDIA Corporation, "CUDA C++ Best Practices," Whitepaper, Mar. 2023. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations`.

[39] A. Grimshaw and D. Merrill, "Parallel scan for stream architectures," 2009.

[40] P. Shirley and S. R. Marschner, *Fundamentals of Computer Graphics (4. ed.)* A K Peters, 2016, ISBN: 9781482229394.