Data-Parallel Flattening by Expansion

Martin Elsman
University of Copenhagen
Denmark
mael@di.ku.dk

Troels Henriksen
University of Copenhagen
Denmark
athas@sigkill.dk

Niels Gustav Westphal Serup
University of Copenhagen
Denmark
ngws@metanohi.name

Abstract

We present a higher-order programmer-level technique for compiling particular kinds of irregular data-parallel problems to parallel hardware. The technique, which we have named "flattening-by-expansion" builds on a number of segmented data-parallel operations but is itself implemented as a higher-order generic function, which makes it useful for many irregular problems. Concretely, the implementation is given in Futhark and we demonstrate the usefulness of the functionality for a number of irregular problems and show that, in practice, the irregular problems are compiled to efficient parallel code that can be executed on GPUs. The technique is useful in any data-parallel language that provides a key set of primitives.

CCS Concepts • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Source code generation**; *Software performance*.

Keywords GPGPU programming, irregular nested parallelism, flattening, functional programming.

ACM Reference Format:
Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. 2019. Data-Parallel Flattening by Expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3315454.3329955

1 Introduction

While the development in computer architectures is increasingly providing improved parallel performance characteristics, the world's programmers have difficulties making efficient use of the increased number of computational parallel units. For some domains, program abstractions make it possible for programmers to make use of libraries to release the potential of new hardware, while such an approach can be difficult to apply for other domains where high performance can be achieved only by parallelising particular domain-specific algorithms. Ideally, one could hope that existing code for such algorithms could be compiled unchanged to exploit the unreleased power of the new architectures. However, although such an approach works sometimes, in general, it does not. In such cases, programmers will often need to master parallel programming using low-level abstractions, such as those provided by CUDA and OpenCL, for which the learning curve is steep and the maintenance and development costs are high.

Modern hardware favors regular data-parallel patterns and it is well understood how also nested regular patterns can be transformed into flat regular parallelism. However, nested irregular parallelism introduces problems that are not easily dealt with. In particular, the overhead of managing segment descriptors (i.e., data that describes the irregularity) and the additional overhead of applying segmented operations become problematic, and, often, the overhead becomes difficult for programmers to understand and reason about.

In 1990, Blelloch introduced the functional programming language NESL [5], a high-level first-order parallel functional language, centered around the idea that nested parallelism (and also irregular parallelism) could be eliminated at compile time by a transformation called *flattening*. NESL was built around a single parallel array comprehension construct, which, as it turned out, could be used to implement a series of parallel constructs such as `map`, `reduce`, `filter`, and `scan` [4, 6–8]. This flattening approach to supporting irregular parallelism has later been refined for efficiency purposes [2, 10, 11, 24]. However, in general, it is difficult for the general flattening techniques to compete with hand-optimised flattened code.

Futhark is a statically typed parallel functional array language, which supports well a notion of so-called *moderate flattening*, which allows for many cases of regular nested parallelism to be mapped to efficient flat parallelism [19]. Whereas regularly nested parallel map constructs can be translated trivially to flat parallelism, it is not immediately clear, in general, how to support efficiently non-regular nested map constructs. Futhark further implements a notion of *incremental flattening* [20], which generates multiple code versions in situations where the most optimal flattening strategy depends on the properties of the input data. For instance, the most optimal GPU code for dense matrix-multiplication depends highly on the sizes of the matrix dimensions. For dealing with irregular problems, Futhark requires the programmer to implement the flattening strategy by hand, which can be quite cumbersome. However, the programmer may choose different strategies for the implementation, which in some cases could involve padding (not work efficient but sometimes the most performance efficient technique in practice) or full flattening, which leads to work efficient implementations, which, however, are sometimes slow in practice.[1]

In this paper, we present a design pattern for obtaining a full-flattened implementation of a certain class of irregular data-parallel problems. The design pattern is implemented in Futhark as a generic higher-order function `expand`, which has the following generic type:

```
val expand 'a 'b : (a → i32) → (a → i32 → b)
                   → []a → []b
```

The function expands a source array, of type `[]a`, into a target array, of type `[]b`, given (1) a function that determines, for each source element, how many target elements it expands to and (2) a function that computes a particular target element based on a source element and the target element index associated with the source. As a simple example, the expression `expand (\x→x) (*) [2,3,1]` returns the array `[0,2,0,3,6,0]`. Here `(\x→x)` denotes the identity function and `(*)` denotes integer multiplication. Semantically, `expand f g xs` performs the equivalent of

```
flatten (map (\x → map (g x) (iota (f x))) xs)
```

where `iota n` produces the array of integers from 0 up to `n-1`, and `flatten` flattens an array of dimension $n+1$ into an array of dimension n. Notice that the inner `map` operates on an array of `f x` elements, which is variant to the outermost `map`. Thus, the array passed to `flatten` is potentially irregular. The purpose of `expand` is to support this kind of irregular map nests in languages that do not directly support irregular parallelism. Given the usual definition of a fused implementation of `flatten` and `map`, called `flatMap`, which has type `(a → []b) → []a → []b`, the semantics of `expand` can also be given by the equation

```
expand f g = flatMap (\x → map (g x) (iota (f x)))
```

As an example of using the `expand` function to solve an irregular problem, consider the task of finding the points in a 2d plane that constitute an array of line segments, each given by its end points; see Figure 1 for an example of a grid of lines. The technique we will use to "draw lines" resembles the development by Blelloch [5] with the difference that it makes use of the `expand` function and that the underlying language implementation does not support irregular parallelism. Using the expand function, all we need is to provide (1) a function that determines for a given line, the number of points that make up the line and (2) a function that determines the n'th point of a particular line, given the index n. The code for such an approach is listed in Figure 2.

The function `points_in_line` makes use of the observation that the number of points that make up the constituting set of points, for the line with end points (x_1, y_1) and (x_2, y_2), is one plus the maximum of $|x_2 - x_1|$ and $|y_2 - y_1|$, that is, one plus the maximum of the absolute values of the difference in x-coordinates and y-coordinates, respectively.[2] Using this observation, the function `get_point_in_line` can independently compute the i'th point in the line by first calculating the proper direction and slope of the line (the two utility functions), relative to the line's starting point. A conditional expression guides whether the x-dimension or the y-dimension is dominating.

Using the flattening-by-expansion approach, we obtain a work efficient implementation of line drawing with work and span complexity being bounded by the work and span complexity of the underlying complexity properties of `scan`, which the implementation of `expand` is founded on.[3] In the concrete case, the two function arguments passed to `expand` are constant time operations, which means that, with n being the number of resulting points, the work complexity of the

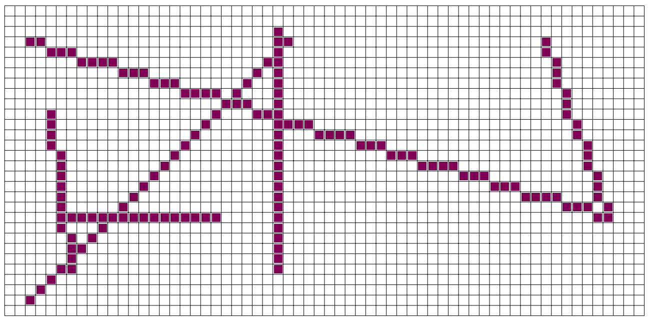

Figure 1. A grid of lines. Each line is defined by its end points and the number of points that make up a line can be determined based on the maximum of the distances between the x coordinates and the y coordinates of the end points.

[1]A parallel algorithm is said to be *work efficient* if the work (i.e., number of operations) performed by the algorithm is of the same asymptotic complexity as the work performed by the best known sequential algorithm that solves the same problem.

[2]In Futhark, the dot-notation is overloaded and used for tuple- and record-projection, module access, and for locally opening a module inside the parentheses following the dot.

[3]By *work* we refer to the total number of operations performed and by *span* we refer to the length of the longest chain of dependent parallel operations (also sometimes called depth.

```
type point = (i32,i32)
type line = (point,point)

let points_in_line ((x1,y1),(x2,y2)) =
  i32.(1 + max (abs(x2-x1)) (abs(y2-y1)))

let compare (v1:i32) (v2:i32) : i32 =
  if v2>v1 then 1 else if v1>v2 then -1 else 0

let slope (x1,y1) (x2,y2) : f32 =
  if x2==x1 then if y2>y1 then 1 else -1
  else r32(y2-y1) / f32.abs(r32(x2-x1))

let get_point_in_line ((p1,p2):line) (i:i32) =
  if i32.abs(p1.1-p2.1) > i32.abs(p1.2-p2.2)
  then let dir = compare p1.1 p2.1
       let sl = slope p1 p2
       in (p1.1+dir*i,
           p1.2+t32(f32.round(sl*r32 i)))
    else let dir = compare p1.2 p2.2
         let sl = slope (p1.2,p1.1) (p2.2,p2.1)
         in (p1.1+t32(f32.round(sl*r32 i)),
             p1.2+i*dir)
let points_of_lines (ls: []line) : []point =
  expand points_in_line get_point_in_line ls
```

Figure 2. Code for drawing lines in parallel. We make use of the truncation function `t32 : f32 -> i32`.

algorithm is $O(n)$ and the span complexity of the algorithm is $O(\log n)$.

The contributions of this paper are the following:

1. We present a generic approach to implementing a class of irregular (and even nested) data-parallel problems using a language-agnostic construct called expand.
2. We present an implementation of expand in Futhark, based on low-level segmented operations.
3. We demonstrate the usefulness of the flattening-by-expansion technique by showing that it can be used for flattening a variety of real-world problems.
4. We demonstrate that the implementation of expand leads to efficient implementations of problems in practice by comparing the performance of the implementations with hand-flattened code in some cases.
5. We discuss the limitations of the approach and give an example of an extension of expand, called expand_reduce, which can be used for implementing, for instance, sparse matrix-vector multiplication.

The remainder of the paper is organised as follows. In the following sections, we present the details of the implementation of the expand function and the underlying segmented operations that the implementation builds on. In Section 4, we demonstrate how the expand function can be used for implementing nested irregular parallelism. In particular, we show how it can be used to expand an array of triangles or circles into an array of lines, which can then be further expanded into an array of points. In Section 5, we show how we can implement a work-efficient data-parallel implementation of Eratosthenes' sieve and in Section 6, we show how the approach can be used to implement sparse matrix-vector multiplication. In Section 7, we show how the technique can be combined with some of Futhark's more elaborate reduction constructs for controlling the depth of objects when drawing graphics. In Section 8, we describe related work and in Section 9, we describe future work and conclude.

All code for the presented examples are available at https://github.com/diku-dk/futhark-array19.

2 A Toolbox of Segmented Operations

Futhark features a number of low-level data-parallel constructs, including map, reduce, scan, and filter. Futhark is a *sequentialising* compiler in that in case some of the parallel constructs may turn up inside already parallel constructs, the compiler is permitted to sequentialise them if it judges this will result in the most efficient code. Futhark implements a number of fusion and flattening transformations that will seek to get as good a performance as possible while maintaining the semantics of the program.

Futhark also features good abstraction mechanisms including higher-order functions [21], polymorphism, record types, and higher-order modules [14], which are all features that are present in the source language, but eliminated at compile time with the aim of obtaining efficient target code.

2.1 Segmented Scan

A key operation needed for working with irregular problems is a segmented scan operation. Whereas specialised segmented scan implementations exist, in Futhark, a segmented scan operation can be defined using the ordinary scan function. Following Blelloch [5, Section 13.2], a generic segmented scan operation can be implemented as follows:

```
let segm_scan [n] 't (op: t → t → t) (ne: t)
              (flags: [n]bool)
              (as: [n]t) : [n]t =
  zip flags as
  |> scan (\(x_flag,x) (y_flag,y) →
           (x_flag || y_flag,
            if y_flag then y else x `op` y))
     (false, ne)
  |> unzip |> (.2)
```

The first to notice about the Futhark implementation of segm_scan is that it is parametric in the type of elements and that Futhark also allows for specifying, using a so-called size-parameter, that the two array arguments should have the same size (i.e., n) and that the resulting array also has size n. This simple support for certain kinds of dependent typing helps the compiler eliminate a number of dynamic checks while at the same time allowing the programmer to specify simple contractual properties of functions. As we shall see later, size parameters may also be referenced as ordinary variables of type i32 in the body of the function, which often makes it straightforward to refer to the size of an argument array.

Given a binary associative operator op with neutral element ne, the function computes the inclusive prefix scan of the segments of as specified by the flags array, where **true** starts a segment and **false** continues a segment. It is a straightforward exercise to prove that, given op is an associative operator with neutral element ne, the function argument passed to scan is an associative operator and (**false**,ne) is its neutral element.

Futhark implements arrays of records (or tuples) as records (or tuples) of arrays, which means that source language zip and unzip operations are compiled into the identity function, which has zero overhead. The higher-order infix "pipe operator" |> passes the result of its left-hand side into the function to the right.

2.2 Replicated Iota

Based on the segm_scan function, we will now present an important utility function called repl_iota. Given an array of natural numbers, specifying repetitions, the function returns an array of weakly increasing indices (starting from 0) and with each index repeated according to the entry in the repetition array. As an example, repl_iota [2,3,1,1] returns the array [0,0,1,1,1,2,3]. The function is defined in terms of other parallel operations, including scan, map, iota, scatter, replicate, reduce, and, as mentioned, segm_scan.

Futhark's scatter function is specified as follows:

```
val scatter 't : *[]t → []i32 → []t → *[]t
```

The first array argument is modified inplace with an association list of updates specified by the following two arrays. The modified array is then transferred back to the caller of scatter. Notice that the uniqueness typing (the *'s in the type), functions here as a simple ownership transfer mechanism.

Here is the definition of repl_iota:

```
let repl_iota [n] (reps:[n]i32) : []i32 =
  let s1 = scan (+) 0 reps
  let s2 = map (\i → if i==0 then 0
                     else unsafe s1[i-1]) (iota n)
  let tmp =
       scatter (replicate (reduce (+) 0 reps) 0)
               s2 (iota n)
  let flags = map (>0) tmp
  in segm_scan (+) 0 flags tmp
```

Whereas the binding of s1 results in an inclusive scan of the repetition values, the binding of s2 results in an exclusive scan. Using the tmp array, which will be of size equal to the resulting array, the flags array will contain **true** values in positions where the indexes should be increased (and zeros elsewhere). The final segmented scan operation will return the desired result.

Notice that in order to use this Futhark code with futhark opencl, we need to prefix the array indexing in line 4 with the **unsafe** keyword; the reason is that Futhark is not sufficiently smart to convince itself that the array indexing in line 4 is always within bounds.

An example evaluation of a call to the function repl_iota is provided in the following table:

Arg/result										
reps	=	[	2	3	1	1	]			
s1	=	[	2	5	6	7	]			
s2	=	[	0	2	5	6	]			
replicate	=	[	0	0	0	0	0	0	0	]
tmp	=	[	0	0	1	0	0	2	3	]
flags	=	[	0	0	1	0	0	1	1	]
segm_scan	=	[	0	0	1	1	1	2	3	]

The resulting array is shown in the last line.

2.3 Segmented Iota

Another useful utility function is the function segm_iota, which, when given an array of flags (i.e., booleans), returns an array of catenated index sequences, each of which is reset according to the booleans in the array of flags. As an example, the expression

```
segm_iota [false,false,false,true,false,false]
```

returns the array [0,1,2,0,1,2]. The segm_iota function can be implemented with the use of a simple call to segm_scan followed by a call to map:

```
let segm_iota [n] (flags:[n]bool) : [n]i32 =
  segm_scan (+) 0 flags (replicate n 1)
  |> map (\x → x-1)
```

The map function call is necessary because segm_scan implements a segmented inclusive scan (contrary to a segmented exclusive scan for which each segment is initiated with an occurrence of the neutral element). Notice that the size-parameter n helps specifying that the size of the result array is of the same size as the given flags array.

3 The Expand Function

Using the utility functions defined in the previous section, we can now define the expand function. The function is listed in Figure 3. The function makes use of the two utility functions

```
let expand 'a 'b (sz: a → i32) (get: a → i32 → b)
                 (arr:[]a) : []b =
  let szs = map sz arr
  let idxs = repl_iota szs
  let iotas = segm_iota (map2 (!=) idxs
                              (rotate (-1) idxs))
  in map2 (\i j → get (unsafe arr[i]) j)
                idxs iotas
```

Figure 3. The definition of the expand function.

repl_iota and segm_iota, which were both presented in Section 2. Assuming that sz and get are constant functions, the dominating function calls of expand are the segmented scan operations appearing inside repl_iota and segm_iota. These calls operate on data of the size $M = \sum_{x \in a}$ sz x, where a is the array argument passed to expand. Under the assumptions of sz and get being constant functions, the work and span complexity of expand is therefore $O(M)$ and $O(\log M)$, respectively.

The expand function can be defined in any parallel language that provides a suitable small set of primitives, namely map, segmented prefix sum (or simply a scan that supports an arbitrary operator, as in Futhark), scatter, and gather. Notice that support for nested parallelism is not required.

3.1 Algebraic Properties and Fusion

The introduced expand function features a number of algebraic properties. We have already presented the semantics of expand in terms of iota, map, and flatten (and an alternative specification in terms of flatMap, which is also sometimes called concatMap).

Another simple algebraic property, which can be used to convert a program into using expand (if that is desired), is the following:

```
expand (const 1) f = map (\x → f x 0)
```

Regular uses of expand can be converted into a regular map-nest using the following algebraic property:

```
expand (const n) f xs =
  flatten <-< map (\i → map (\x → f x i) xs)
                (iota n)
```

Here the infix Futhark function <-< denotes function composition. Futhark does not currently recognise such patterns as expand is a user defined function. Futhark will, however, happily inline the const function inside the expand function at every use of expand. This inlining could potentially give rise to optimisations, which, however, are not currently exploited.

A proper fusion scheme is essential for any language that targets GPUs [10, 16, 25, 29]. Futhark implements a number of fusion strategies but is also careful not to introduce duplication of work [18].

The expand function fuses with map and filter as follows:

```
map f <-< expand sz get =           -- map
  expand sz (\x → f <-< get x)      -- fusion

expand sz get <-< filter p =        -- filter
  expand (\x → if p x then sz x     -- fusion
              else 0) get
```

Because Futhark supports well map-map fusion [18] and because applications of expand are inlined by Futhark, essentially, expand fuses with map. Fusing expand with filter, however, is not easily supported, however, unless the Futhark fusion engine gets to learn about the intrinsic fusion properties of expand.

4 Nested Irregular Parallelism

In this section, we demonstrate that the flattening-by-expansion technique can also be applied in a nested setting with flattening happening at multiple levels.

4.1 Drawing Triangles

An example of an algorithm worthy of flattening is triangle rasterisation, that is, an algorithm that in parallel computes the points that constitute a set of triangles. The algorithm that we present here is based on the assumption that we already have a function for drawing multiple horizontal lines in parallel. Luckily, we have already seen how we can define such a function! The algorithm for drawing triangles is based on the property that any triangle can be split into an upper triangle with a horizontal baseline and a lower triangle with a horizontal ceiling. Just as the algorithm for drawing lines makes use of the expand function defined earlier, so will the flattened algorithm for drawing triangles. A triangle is defined by the three points representing the corners of the triangle:

```
type triangle = (point, point, point)
```

We shall make the assumption that the three points that define the triangle have already been sorted according to the y-axis. Thus, we can assume that the first point is the top point, the third point is the lowest point, and the second point is the middle point (according to the y-axis).

The first function we need to pass to the expand function is a function that determines the number of horizontal lines in the triangle:

```
let lines_in_triangle ((p,_,r):triangle) : i32 =
  r.2 - p.2 + 1
```

The second function we need to pass to the `expand` function is somewhat more involved. We first define a function `dxdy`, which computes the inverse slope of a line between two points:[4]

```
let dxdy (a:point) (b:point) : f32 =
  let dx = b.1 - a.1
  let dy = b.2 - a.2
  in if dy == 0 then 0
     else r32 dx / r32 dy
```

We can now define the function that, given a triangle and the horizontal line number in the triangle (counted from the top), returns the corresponding line:

```
let get_line_in_triangle ((p,q,r):triangle)
                         (i:i32) =
  let y = p.2 + i in
  if i <= q.2 - p.2 then     -- upper half
    let sl1 = dxdy p q
    let sl2 = dxdy p r
    let x1 = p.1+t32(f32.round(sl1*r32 i))
    let x2 = p.1+t32(f32.round(sl2*r32 i))
    in ((x1,y),(x2,y))
  else                       -- lower half
    let sl1 = dxdy r p
    let sl2 = dxdy r q
    let dy = (r.2 - p.2) - i
    let x1 = r.1-t32(f32.round(sl1*r32 dy))
    let x2 = r.1-t32(f32.round(sl2*r32 dy))
    in ((x1,y),(x2,y))
```

The function distinguishes between whether the line to compute points for resides in the upper or the lower subtriangle. Finally, we can define a parallel, work-efficient function that converts a number of triangles into lines:

```
let lines_of_triangles (xs:[]triangle) : []line =
  expand lines_in_triangle get_line_in_triangle
         (map normalise xs)
```

In the above function, the function `normalize` sorts (using an unrolled bubble sort) the corner points in each triangle according to the y-axis:

```
let normalise ((p,q,r): triangle) : triangle =
  let bubble (a:point) (b:point) =
    if b.2 < a.2 then (b,a) else (a,b)
  let (p,q) = bubble p q
  let (q,r) = bubble q r
  let (p,q) = bubble p q
  in (p,q,r)
```

[4]For converting floats to integers, we make use of the function `r32 : i32 -> f32`.

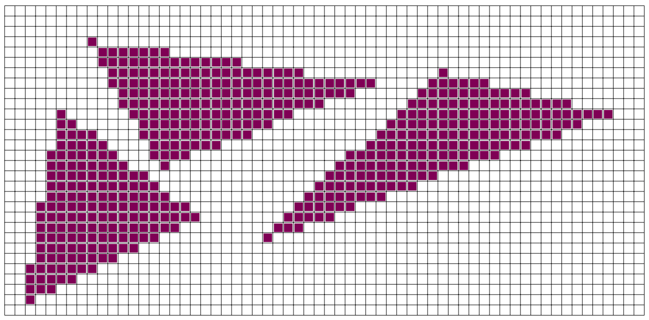

Figure 4. A grid of points generated by first, in parallel, generating the lines that make up a number of triangles, and then, also in parallel, generating the points that make up the lines. The entire algorithm is work-efficient due to flattening and the use of the `expand` function.

Figure 4 shows the code in action when the function `lines_of_triangles` is called with an array of three triangles, defined as follows:

```
[((5,10), (2,28) , (18,20)),
 ((42,6), (58,10), (25,22)),
 ((8,3) , (15,15), (35,7))]
```

The lines generated by the function `lines_of_triangles` is further processed using the `points_of_lines` function, which generates the points that are then shown in a grid of height 30 and width 62.

The technique demonstrated for triangles can easily be adapted to work for solid circles and ellipses. The technique can also be adapted to work for drawing the circumference of regular polygons and circles.

5 Flattening the Sieve of Eratosthenes

A sometimes useful strategy for obtaining a parallel algorithm is to use the concept of contraction, the general algorithmic trick of solving a particular problem by first making a contraction step, which simplifies the problem size, and then repeating the contraction algorithm until a final result is reached [27]. A variant of a contraction algorithm is an algorithm that first solves a smaller problem, recursively, and then uses this result to provide a solution to the larger problem. One such algorithm is a version of the Sieve of Eratosthenes that, to find the primes smaller than some n, first calculates the primes smaller than $\sqrt{n}$. It then uses this intermediate result for sieving away the integers in the range $\sqrt{n}$ up to n that are multiples of the primes smaller than $\sqrt{n}$.

Unfortunately, Futhark does not presently support recursion, thus, one needs to use a `loop` construct instead to implement the sieve. A Futhark program calculating an array containing the set of primes below some number n, is shown in Figure 5.

```
let primes (n:i32) : []i32 =
  (.1) <|
  loop (acc,c) = ([],2) while c < n+1 do
    let c2 = if c < t32(f32.sqrt(r32(n+1)))
             then c*c
             else n+1
    let is = map (+c) (iota(c2-c))
    let fs = map (\i →
                   let xs = map (\p → if i%p==0
                                       then 1
                                       else 0) acc
                   in reduce (+) 0 xs) is
    -- apply the sieve
   let new = filter (\i → 0 == unsafe fs[i-c]) is
   in (acc ++ new, c2)
```

Figure 5. Non-flattened version of the Sieve of Eratosthenes.

Notice that, when computing `c2`, we are careful not to introduce overflow by not calculating `c*c` unless `c` is less than the square root of `n+1`. Notice also that the algorithm applies a parallel sieve for each step, using a combination of maps and reductions. The best known sequential algorithm for finding the number of primes below some n takes time $O(n \log \log n)$. Although the present algorithm is quite efficient in practice, it is not work efficient, since the work inside the loop is superlinear. The loop itself introduces a span with a $\log \log n$ factor because the size of the problem is squared at each step, which is identical to doubling the exponent size at each step (i.e., the sequence $2^2, 2^4, 2^8, 2^{16}, \ldots, n$, where $n = 2^{2^m}$, for some positive m, has $m = \log \log n$ elements.)

We now present a work-efficient variant of the above function for computing primes using the concept of flattening-by-expansion. The function is listed in Figure 6.

There are two points to notice about the code. First, we use the `expand` function to calculate and flatten the sieves. For each prime `p` and upper limit `c2`, we can compute the number of contributions in the sieve (the function `sz`). Then, for each prime `p` and sieve index `i`, we can compute the sieve contribution (the function `get`). Second, using a `scatter`, a `map`, and a `filter`, we can compute the new primes in the interval `c` to `c2`. The algorithm presented here is work efficient, has work complexity $O(n \log \log n)$, and span complexity $O(\log \log n)$.

We have compared the two Futhark algorithms running with the OpenCL backend to a straightforward C implementation of the Sieve of Eratosthenes for finding the number of primes below 100,000,000. The experiments were performed on a machine with an NVIDIA RTX 2080 Ti GPU (using `futhark opencl`) and an Intel Xeon E5-2650 running at 2.60GHz (using `futhark c`). We report timing results as averages over 10 runs. The plain C version takes 530ms on average, the non-flattened version takes 171ms on average,

```
let primes (n:i32) =
  (.1) <|
  loop (acc:[]i32,c) = ([],2) while c < n+1 do
    let c2 = if c < t32(f32.sqrt(r32(n+1)))
             then c*c
             else n+1
    let sz (p:i32) = (c2 - p) / p
    let get p i = (2+i)*p
    let sieves : []i32 = map (\p → p-c)
                              (expand sz get acc)
    let vs = replicate (c2-c) 1
    let vs = scatter vs sieves
                     (replicate (length sieves) 0)
    let new = filter (>0) <| map2 (*) vs (c..<c2)
    in (acc ++ new, c2)
```

Figure 6. Flattened version of the Sieve of Eratosthenes using flattening-by-expansion.

and the flattened version, which uses the `expand` function, takes 11.3ms on average. We emphasise here that we have arranged that the versions are comparable in the sense that they all compute the sieves from scratch.

For these and later measurements, we do not measure the time taken to move input data to the GPU, or results back to the CPU. The operations in this paper tend to be building blocks in larger Futhark programs, not full applications, and it is our experience that in practice, their data is already located on the GPU, and their results also need further processing on the GPU.

6 Sparse Matrix-Vector Multiplication

Numerous possible representations of sparse matrices exist. Here we demonstrate the use of the flattening-by-expansion technique for implementing a version of sparse-matrix vector multiplication based on a compressed sparse row implementation of sparse matrices. In Futhark, the type of a compressed sparse row representation of a matrix can be defined as follows:

```
type csr 't = {row_off: []i32,
               col_idx: []i32,
               vals: []t}
```

The type `csr` is parameterised over the type of the underlying matrix values. Given a sparse matrix of size $N \times M$ with NNZ non-zero values, the size of the `row_off` array is $N + 1$ and the size of each of the `col_idx` and `vals` arrays is NNZ. The compressed sparse row representation favours that each row can be processed in parallel. However, because each row contains a different number of non-zero elements, the problem becomes irregular. We shall apply an extended version of the expand function, which has the following type:

```
let smvm ({row_off,col_idx,vals} : csr f32)
         (v:[]f32) : []f32 =
  let rows = map (\i → (i,
                        row_off[i],
                        row_off[i+1]-row_off[i]))
                 (iota(length row_off - 1))
  let sz r = r.3
  let get r i = vals[r.2+i] * v[col_idx[r.2+i]]
  in expand_reduce sz get (+) 0f32 rows
```

Figure 7. Flattened implementation of sparse matrix-vector multiplication in Futhark.

```
val expand_reduce 'a 'b [n] : (a → i32)
                           → (a → i32 → b)
                           → (b → b → b) → b → [n]a → [n]b
```

The `expand_reduce` function makes use of the internal flags array (defined inside `expand`) to call a function `segm_reduce` with the arguments of type `(b → b → b)` and `b` to reduce the expanded values. The function `segm_reduce` can be defined in terms of the `segm_scan` operation, but we shall not give the definition here.

Using the `expand_reduce` function, sparse matrix-vector multiplication can be defined as shown in Figure 7. The function `smvm` expands each row into an irregular number of multiplications and then reduces these multiplication results using the monoid defined by the associative function `(+)` and its neutral element `0`. Letting `smvm` be parameterised over multiplication and the monoid structure would allow us to instantiate the `smvm` function to work on, for instance, matrices and vectors containing `f64` values.

The following table shows the performance of multiplying different sparse matrices of dimension $10,000 \times 10,000$ with a dense vector of size $10,000$. Again, the experiments were performed on a machine with an NVIDIA RTX 2080 Ti GPU (using `futhark opencl`) and an Intel Xeon E5-2650 running at 2.60GHz (using `futhark c`). We report timing results (averages over 10 runs) for densities of 1-20 percent:

Density	Futhark C (ms)	Futhark OpenCL (ms)
1%	27.1	0.52
2%	49.9	0.79
3%	74.1	1.10
4%	97.0	1.40
5%	130.3	1.70
10%	264.6	3.12
15%	381.2	4.55
20%	506.6	5.90
Dense	**94.01**	**2.77**

The sparse matrix density (reported in percentages) corresponds to the number of non-zero elements (in millions). The last entry shows numbers for a dense matrix-vector multiplication, and we see that when the density gets higher than approximately 10 percent, dense matrix-vector multiplication outperforms the sparse version. This is because of lower constant factors; for the dense matrix-vector multiplication, the Futhark compiler generates a transposition to ensure coalesced memory access, followed by a call to a single GPU kernel that performs the actual computation. In contrast, the sparse operation requires several expensive `scans`.

6.1 Sparse Matrix-Matrix Multiplication

It turns out that it is straightforward to implement sparse matrix-matrix multiplication on top of the functionality already developed. Here is a function that implements multiplication of a sparse matrix with a dense matrix:

```
let smmm [n] (sm:csr f32) (m:[][n]f32) :[n][]f32 =
  map (smvm sm) (transpose m)
```

It is more difficult to implement matrix multiplication between two sparse matrices, for which efficient implementations require some degree of binary searching.

7 Managing Graphics Depth

The triangle drawing technique presented in Section 4.1 works only when all triangles have the same color. In essence when two triangles overlap, the lines, and eventually the points, generated with `lines_of_triangles` and `points_of_lines` can be used in concert with `scatter` to draw the triangles on a canvas. However, `scatter` does not provide any guarantees about the effect of writing multiple values to the same entry, except when the values are identical.

To deal with this problem, Futhark features a function called `reduce_by_index`, which can be used instead of `scatter` to control the effect of multiple writes to the same entry. Here is the type of the function:

```
val reduce_by_index 'a : *[]a → (a → a → a) → a
                       → []i32 → []a → *[]a
```

In addition to the array arguments, also taken by `scatter`, `reduce_by_index` also takes an (assumed to be) associative and commutative function, operating on array elements, and its neutral element. This function is used to combine the old value in the array with the new one that is being written. We shall not discuss the implementation of `reduce_by_index` here but just mention that its implementation is based on the techniques used for implementing histogram computations on GPUs [23, 26]. In fact, `reduce_by_index` can be viewed as a generalised function for computing histograms.

Using a 3d representation of colored points, lines, and triangles, we can now make use of `reduce_by_index` to control which parts of triangles are shown. We do this by pairing each pixel to be written with its distance from the camera,

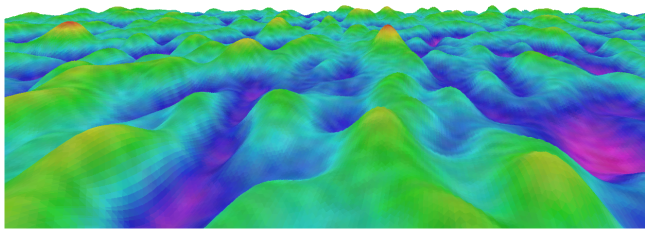

Figure 8. A landscape scene from a game implemented in Futhark. The terrain is displayed using a large number of triangles, which are colored according to their vertical positions in the terrain.

and providing an operator to `reduce_by_index` that picks the pixel closest to the camera. This technique is exactly the classic technique of z-buffering.

Figure 8 shows a scene from a game implemented in Futhark, where the landscape is drawn using a set of triangles, colored differently based on the vertical position in the terrain.

Using the AMD Radeon Pro 460 GPU on a MacBook Pro,[5] 500,000 triangles can be drawn using flattening-by-expansion with a frame rate of 15 frames per second (on a 2880×1800 display). These numbers include the time for computing the triangles, based on the camera's point-of-view and the terrain information, and for copying the computed images back and forth between the GPU and CPU (Futhark cannot presently store images directly in image buffers). Notice that this implementation is of course much less efficient than if we used the specialised graphics hardware on the GPU. We do not claim competitive rendering performance; merely that `expand` allows us to express the algorithm in a natural and parallel way, and still obtain decent performance.

8 Related Work

Much related work has been carried out in the area of supporting nested parallelism, including the seminal work on flattening of nested parallelism in NESL [5, 6], which was extended to operate on a richer set of values in Data-parallel Haskell [9], and the work on data-only flattening [34]. These approaches tend to focus on maximising expressed parallelism, and negate the need for a function such as `expand`. However, compiler-based flattening has proven challenging to implement efficiently in practice, particularly on GPUs [3]. Other promising attempts at compiling NESL to GPUs include Nessie [28], which is still under development, and CuNesl [34], which aims at mapping different levels of nested parallelism to different levels of parallelism on the GPU, but lacks critical optimisations such as fusion.

[5]In contrast to the compute servers we are using for the other benchmarks, a MacBook Pro has a screen.

More recent data-parallel languages include Obsidian [12, 30, 31] and Accelerate [10], which are both embedded in Haskell, and do not feature arbitrary nested parallelism. Accelerate in particular can easily support manually flattened programming in the `expand` style, as segmented scans and scatter operations are readily available. Accelerate also supports certain forms of irregular arrays by supporting a notion of irregular stream scheduling [13].

Other attempts at supporting nested (and even irregular) parallelism on GPUs include more dynamic approaches, such as dynamic thread block launching [33] and dynamic parallelism, which are extensions to the GPU execution model involving runtime and micro architecture changes. These approaches to supporting irregular parallelism does, however, often come with a significant overhead [32]. Other dynamic approaches include a partial flattening approach, implemented using thread stealing, which also introduce a significant overhead [22].

9 Conclusions and Future Work

In this paper, we have demonstrated a programming technique that allows for convenient manual flattening of certain irregular nested parallel constructs, even if the target language does not support nested parallelism at all. The resulting code is asymptotically as efficient as that which would have been generated with full NESL-style flattening, and allows the programmer more control and a "pay-as-you-go strategy" to flattening. Further, the real-world performance is sufficient to carry out real-time graphics rendering.

There are a number of possibilities for future work. First, some overhead can perhaps be avoided in situations where flattened data is immediately scattered into a target array. To avoid the resulting double copying overhead, one may consider defining a function that instead of returning a target array takes as argument a destination array, which is then returned to the caller with modified content. Second, there are a number of irregular nested parallel algorithms that may benefit from the use of the `expand` function. Such algorithms include algorithms for graph traversals [17] and irregular parallel financial applications [1].

Other possible future work include investigating whether the technique can be extended in such a way that it can be used to ease the flattening of more involved algorithms, such as quick-sort [15] or multiplication of two sparse matrices.

Acknowledgments

This research has been partially supported by the Independent Research Fund Denmark grant under the research project *FUTHARK: Functional Technology for High-performance Architectures.* Thanks to the reviewers for their careful comments and to Cosmin E. Oancea for many fruitful discussions about this work.

References

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages.

[2] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-only Flattening for Nested Data Parallelism. In *Procs. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/2442516.2442525

[3] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the Gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/2364527.2364563

[4] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

[5] Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge.

[6] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM (CACM)* 39, 3 (1996), 85–97.

[7] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. ACM, New York, NY, USA, 213–225. https://doi.org/10.1145/232627.232650

[8] Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing* 21, 1 (1994), 4–14.

[9] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*. 10–18.

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

[11] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. 2008. Partial vectorisation of Haskell programs. In *Proc ACM Workshop on Declarative Aspects of Multicore Programming, San Francisco*.

[12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Work. on Decl. Aspects of Multicore Prog DAMP*. 21–30.

[13] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming Irregular Arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 174–185. https://doi.org/10.1145/3122955.3122971

[14] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97 (July 2018), 30 pages.

[15] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. 2018. *Parallel Programming in Futhark*. Department of Computer Science, University of Copenhagen. https://futhark-book.readthedocs.io

[16] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. 2006. With-Loop Fusion for Data Locality and Parallelism. In *Proceedings of the 17th International Conference on Implementation and Application of Functional Languages (IFL'05)*. Springer-Verlag, Berlin, Heidelberg, 178–195. https://doi.org/10.1007/11964681_11

[17] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2018. Modular Acceleration: Tricky Cases of Functional High-Performance Computing. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC '18)*. ACM, New York, NY, USA.

[18] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '13)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2502323.2502328

[19] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[20] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[21] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*.

[22] Ming-Hsiang Huang and Wuu Yang. 2016. Partial Flattening: A Compilation Technique for Irregular Nested Parallelism on GPGPUs. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP '16)*. 552–561. https://doi.org/10.1109/ICPP.2016.70

[23] Wookeun Jung, Jongsoo Park, and Jaejin Lee. 2014. Versatile and Scalable Parallel Histogram Construction. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2628071.2628108

[24] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation Avoidance. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2364506.2364512

[25] Trevor L. McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*.

[26] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. 2011. High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*. ACM, New York, NY, USA, Article 1, 8 pages. https://doi.org/10.1145/1964179.1964181

[27] Course Organizers. 2016. *Algorithm Design: Parallel and Sequential*. Carnegie Mellon University. Course Book Draft Edition. Course Taught Fall 2016 by Umut Acar and Robert Harper.

[28] John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at the *Compilers for Parallel Computing Workshop (CPC '15)*. Imperial College, London, UK.

[29] Amos Robinson, Ben Lippmeier, and Gabriele Keller. 2014. Fusing Filters with Integer Linear Programming. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '14)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2636228.2636235

[30] Joel Svensson. 2011. *Obsidian: GPU Kernel Programming in Haskell*. Ph.D. Dissertation. Chalmers University of Technology.

[31] Joel Svensson, Mary Sheeran, and Koen Claessen. 2011. 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 (IFL'08)*. Springer-Verlag, Berlin, Heidelberg, 156–173. http://dl.acm.org/citation.cfm?id=2044476.2044485

[32] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chitaranjan Das. 2017. Controlled Kernel Launch for Dynamic Parallelism

in GPUs. In *Proceedings of the 2017 IEEE 23rd Symposium on High Performance Computer Architecture, HPCA 2017 (HPCA '17)*. IEEE Computer Society, United States, 649–660. https://doi.org/10.1109/HPCA.2017.14

[33] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2015. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 528–540. https://doi.org/10.1145/2749469.2750393

[34] Yongpeng Zhang and Frank Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP'12)*. IEEE Computer Society, Washington, DC, USA, 340–349.