Strategies for Regular Segmented Reductions on GPU

Rasmus Wriedt Larsen
Department of Computer Science, University of Copenhagen (DIKU)
Denmark
rasmuswriedtlarsen@gmail.com

Troels Henriksen
Department of Computer Science, University of Copenhagen (DIKU)
Denmark
athas@sigkill.dk

CCS Concepts • **Software and its engineering → General programming languages**; • **Theory of computation** → *Program analysis*;

Keywords GPGPU, parallelism, functional programming

ACM Reference format:
Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, Oxford, UK, September 7, 2017 (FHPC'17),* 10 pages.
https://doi.org/10.1145/3122948.3122952

1 Introduction

Massively parallel computers have long since entered the mainstream in the form of GPUs. Yet their use still remains mostly restricted to a small group of experts, who then produce high-performance libraries useful by non-specialist programmers. While useful, such libraries are less flexible and powerful than programming the GPU directly.

The reason for this unfortunate state of affairs must be sought in the low-level nature of the GPU programming APIs, such as CUDA and OpenCL. Fortunately, recent years have seen a rise in work on parallel functional languages [2, 3, 6, 8, 9, 17] that present a more friendly programming model.

The challenge when programming massively parallel computers is not the handling of embarrassingly parallel constructs, like map, but rather in the efficient implementation of those constructs that require some synchronisation and communication, such as scans and reductions. One common operation in parallel programming is reducing each inner array of a multidimensional array, such as rows of a matrix. This is called a *segmented reduction*. For example, we can use this to compute the sum of the rows of a matrix, yielding a vector. We use the term *segment size* to refer to the size of the inner array (the segment) being reduced.

This paper does not claim to provide the final key that unlocks the power of GPUs for the masses, but merely seeks to advance the state-of-the-art for one particular pattern: the regular segmented reduction. In a regular segmented reduction, all segments have the same size. While less flexible than an irregular segmented reduction, this restriction permits a more efficient implementation, and is sufficient for many tasks.

FHPC'17, September 7, 2017, Oxford, UK

ACM ISBN 978-1-4503-5181-2/17/09…$15.00
https://doi.org/10.1145/3122948.3122952

While there are existing libraries and languages supporting regular segmented reductions, they are not fully efficient. Some libraries, such as Thrust [16], do not support regular segmented reductions directly, but do support irregular segmented reductions, which can be used to express regular segmented reductions albeit with some overhead.

Accelerate [6], a Haskell eDSL for parallel programming, supports regular segmented reductions, but always uses one GPU workgroup per segment. This is inefficient for those reductions that involve a few very large segments, or those that involve a large number of small segments. A similar strategy is taken by Delite [17], in which all inner parallelism is mapped at warp or block level.

This paper proposes an *adaptive* approach that chooses an evaluation strategy at runtime based on the segment size and number of segments. The contributions of this paper are:

First, we describe in Section 2 three strategies for handling regular segmented reductions. Each strategy is optimal for a certain class of datasets, which are characterized by the segment size and the number of segments.

Second, we describe in Section 3 a modification of the compiler for Futhark [11–15], a functional array language, that incorporates the three strategies, and switches automatically between them based on runtime information.

Third, we present (also in Section 3) a systematic evaluation of our approach on four synthetic benchmarks that shows that the three strategies efficiently cover all cases of number of segments and segment size. The baseline used in this evaluation is Futhark's non-segmented reduction.[1], as well as implementations using CUB [18]. This is a suitable baseline because, on the one hand, reduction should be more efficient, since it is in essence a "simpler" operation than its segmented counterpart, and, on the other hand, Futhark's reduction [14] is relatively efficient (and has been found to outperform library-based implementations such as Thrust).

Finally, we demonstrate in Section 4 that our approach results in significant (application-level) speedup on two benchmarks ported from the Rodinia [7] benchmark suite (K-means and Backprop).

2 The Three Strategies

The efficient implementation of non-segmented reductions on GPUs is well studied in the literature [10, 14, 19]. A common technique for reducing an input array of size n is to spawn k *workgroups*[2], each consisting of w threads, where each thread sequentially reduces a chunk of the input consisting of $\frac{n}{w \times k}$ elements, producing a per-thread intermediate result, which is then reduced inside each workgroup (using tree reduction on fast memory) to one result per workgroup. If w is picked to be less than the maximum workgroup

[1] Throughout the paper we use *reduction* to refer to general compositions of maps and reductions.
[2] This paper follows the OpenCL terminology. In NVIDIAs CUDA, a workgroup is called a *thread block*.

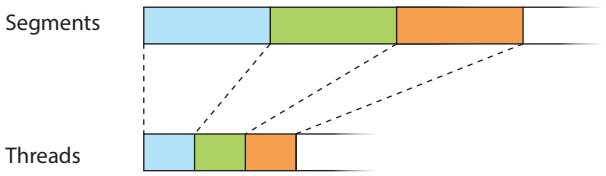


Figure 1. The sequential segments strategy will process each segment sequentially in one thread.

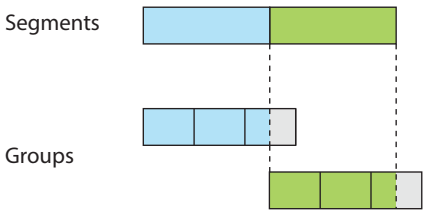


Figure 2. The large segments strategy can use multiple workgroups to reduce a single segment; this will generate multiple intermediate results that will need to be reduced further. In this example we use three workgroups for each of the two segments.

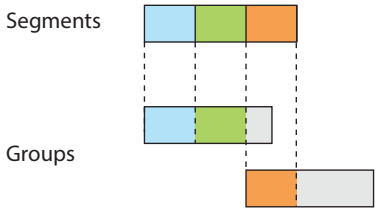


Figure 3. The small segments kernel can process multiple whole segments within a single workgroup. In this example each workgroup can process two whole segments, but there is only one segment to reduce for the last workgroup.

size, we can then launch a new reduction with one workgroup of size w that reduces the w per-workgroup results into the final result of the reduction.

This approach works well for most reduction operators and input sizes (although there are of course many low-level tuning techniques employed to improve the constant factors). Unfortunately, for segmented reductions, no single technique is optimal. One technique that is sometimes used is to recast a segmented reduction as a *segmented scan*, followed by extracting the last element from each of the produced segments. It has been shown that a segmented scan can be implemented as an operator transformation of a corresponding non-segmented scan [4]. Thus, if we have access to an implementation of scan, we can also perform a segmented reduction. However, as we shall see, this implementation performs poorly in practice due to the extra memory traffic imposed by the scan.

Our idea for computing segmented reductions is to use three different kernel strategies to effectively cover all cases of number of segments and segment size:

Sequential segments : When there are enough segments that we can fully utilize the GPU by computing the reduction for each segment sequentially in a single thread, we should do so, because this eliminates inter-thread communication.

Large segments : When the size of a segment is large enough, we can use an approach similar to a non-segmented reduction, where we use one or more (whole) workgroups to perform the reduction of a single segment.

Small segments : When there are so few segments that launching one thread per segment would not saturate the GPU, and the segments are so small that launching one workgroup per segment would not be efficient, we process several (whole) segments per workgroup. This allows an implementation that performs intra-workgroup segmented scan in fast memory, but does not require inter-workgroup communication.

We notice that all three strategies leverage the property that all segments have the same size. For example, this guarantees that the sequential-segment strategy is load balanced, and allows a straightforward mapping between segments and workgroups for the large- and small-segment strategies.

In principle, there is also a fourth option: sequentialising the entire segmented reduction, possibly not involving the GPU at all. This may be the optimal strategy for very small data sets, especially if they are already located on the CPU. However, we shall not go in this direction.

The rest of this section describes the implementation and performance characteristics of the three strategies in detail. We are not as much interested in absolute numbers, as much as how the different strategies perform on different inputs.

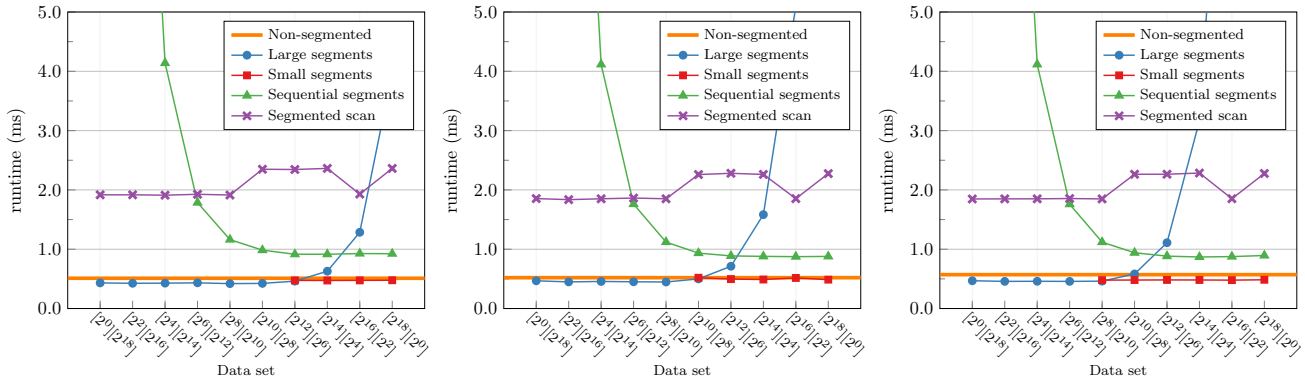
The performance of the different strategies on an NVIDIA K40 GPU is shown on Figure 4, which summarises the performance of a segmented integer summation on two datasets; containing respectively 2^{18} (Figure 4a) and 2^{26} (Figure 4b) integers in total. For each of the two data sets, we show runtime for different decompositions into number of segments and segment size, while the work remains constant. We can thus see how a strategy may be efficient for some input decompositions, but not for others.

We conduct the measurements with three different GPU workgroup sizes to show how this impacts the performance. While this shifts the thresholds at which the different strategies become optimal, it does not change the overall picture. As a point of reference, we also show the performance of a *non-segmented* summation, as well as a segmented summation implemented with a segmented prefix sum.

2.1 Sequential Segments

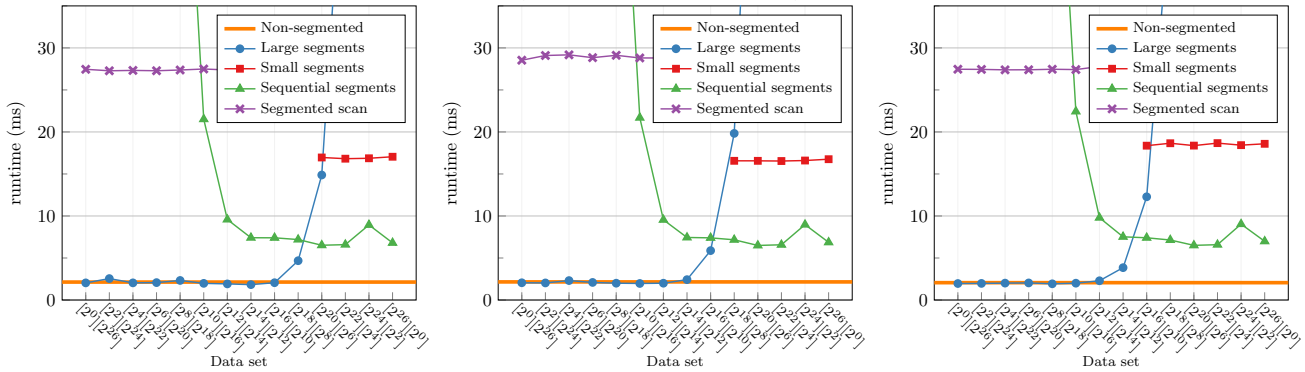
The *sequential segments* strategy (visualised on Figure 1) is to launch one GPU thread for each segment, which then performs a sequential reduction.

The loop-in-map approach has a very straightforward implementation: we launch as many threads as there are segments, and each thread will iterate over the elements of a single segment and apply the reduction operator. A straightforward serial iteration would result in non-coalesced memory accesses. There are several ways to resolve this issue. One common technique is to transform the thread index space, but this is not useful in this case, because each



Workgroup size of 128 threads | Workgroup size of 512 threads | Workgroup size of 1024 threads

(a) Segmented summation of 2^{18} integers.



Workgroup size of 128 threads | Workgroup size of 512 threads | Workgroup size of 1024 threads

(b) Segmented summation of 2^{26} integers.

Figure 4. The runtime of segmented summation using various implementation strategies and workgroup sizes for a different of input sizes. The amount of work done is constant along x-axis; only the ratio between number of segments and segment size changes.

thread *must* read a logically contiguous segment of the input array. Our solution is to transpose the array, such that the segments are stored with a stride in memory. For large data sizes, the cost of this transposition is much less than the penalty of non-coalesced memory accesses.

As we can see on Figure 4, performance is abysmal when we have few large segments, as we fail to provide enough parallelism to the GPU. One interesting detail is that for the small 2^{18} dataset, this strategy becomes more efficient than segmented scan at 2^8 segments, while for the larger dataset this threshold is not crossed until we are processing 2^{10} segments. The reason is that the segments are much larger in the latter case than the former; implying more sequential work.

2.2 Large Segments

The *large segments* strategy (visualised on Figure 2) is effective when the segment size is larger than the workgroup size, as we can then use multiple (whole) workgroups to reduce one segment. This will allow us to use more active threads, than just using a single workgroup per segment – so when there is only a few large segments, this will give us better performance.

We end up producing a number of partial results per segment—one per workgroup used[3]—that have to be reduced further in a second stage. This second segmented reduction will be over the same number of segments, but each will be orders of magnitude smaller than in the original input, so the choice of algorithm is less important (although the small segments strategy is a good choice)

When the segments are very large, then instead of launching thousands of workgroups per segment, we can make each thread of a workgroup sequentially process multiple consecutive elements. This chunking procedure trades excess parallelism for *efficient sequentialisation*. This can improve performance significantly, because multiple memory transactions can be in flight at once. This will also have the added benefit of reducing the number of partial results that have to be reduced in the second stage. For a non-commutative reduction operator, making each thread read multiple elements will require that the input array has been transposed, to achieve memory coalescing.

If the number of segments is so large that it would not increase occupancy to use multiple workgroups per segment, we can use a single workgroup to process an entire segment, thereby also avoiding the cost of the recursive reduction of the partial results.

[3] One workgroup always processes elements from the same segment.

```
1  Input: s -- segment length
2         m -- number of segments
3         w -- number of threads per workgroup
4         p -- number of workgroups
5         A -- the input (global) array of size (m * s)
6         T -- temporary workgroup-local array of length w
7
8  Assumes: w < s, p <= m, and that p*w threads fully utilize hardware.
9
10 Output: B -- array of total length p, semantically
11            -- consisting of m segments of length p/m.
12            -- If p==m then stop else recursively reduce B.
13
14 c ← (m*s) / (w*p)        -- per thread chunking factor
15 forall i in [0...p-1]:    -- for all workgroups
16  forall j in [0...w-1]:   -- for all workgroup's threads
17   tid ← i*w+j -- global thread id
18   T[j] ← thread tid sequentially reduces
19           elements A[tid*c : (tid+1)*c-1]
20  endforall
21
22  B[i] ← workgroup i reduces array T in parallel
23 endforall
```

Figure 5. Large-Segment Strategy Pseudocode. For simplicity, all divisions are assumed to be exact, and the workgroup size divides the segment size.

In essence, large-segment strategy is very similar to the case of non-segmented reduction, which is known to have an efficient implementation. A simplified pseudocode for this strategy is sketched in Figure 5.

From Figure 4 we can see that the large segments strategy has the same runtime as a non-segmented reduction when there are many large segments. When the segments are only a few times larger than the workgroup size, each thread will only process few elements, and therefore the runtime increases; this is why we see the large segments strategy being viable for larger work sizes when using a workgroup size of 128, compared to 1024. In Figure 4a the large segments strategy is a bit faster than the non-segmented reduction, but this is only due to tuning parameters[4].

2.3 Small Segments

The *small segments* strategy (visualised on Figure 3) can be used when the segment size is smaller than the workgroup size. Processing multiple segments inside a single workgroup is more efficient than using one workgroup per segment if the segments are so small that most of the threads of the workgroup would be idle.

To avoid (expensive) inter-workgroup communication, we restrict the implementation to processing a whole number of segments within one workgroup. That is, a segment cannot span two workgroups. Therefore, we might waste some threads because we cannot fill the workgroup completely. The input data will not need to be transposed to ensure coalesced memory accesses, because consecutive threads already read consecutive elements. A simplified pseudocode for this strategy is sketched in Figure 6.

[4] changing the number of workgroups used by the non-segmented reduction would make it as fast

```
1  Input: s -- segment length
2         m -- number of segments
3         w -- number of threads per workgroup
4         p -- number of workgroups
5         A -- the input (global) array of size (m * s)
6         T -- temporary workgroup-local array of length w
7
8  Assumes: w > s and s*m == w*p
9
10 Output: B -- array of total length m
11
12 sgm_per_wgroup ← w / s
13 forall i in [0...p-1]:    -- for all workgroups
14  forall j in [0...w-1]:   -- for all workgroup's threads
15   tid ← i*w+j             -- global thread id
16   T[j] ← A[tid]
17  endforall
18
19  T ← workgroup i performs a parallel segmented scan of array T
20
21  forall j in [0...w/s-1]: -- write the result to global memory
22   B[(w/s)*i+j] = T[j*s]
23  endforall
24 endforall
```

Figure 6. Small-Segment Strategy Pseudocode. For simplicity, all divisions are assumed to be exact, and the segment size divides the workgroup size.

From Figure 4 we can see that the small segments strategy is only a good choice for small input sizes. This is partially because it is able to exploit all available parallelism, and partially because there is no need to transpose the input to obtain coalesced memory accesses. On larger data sets, the added overhead and intra-workgroup synchronisation outweighs these benefits, and the sequential segments strategy is superior. This is also because the small segments strategy cannot benefit from efficient sequentialisation via chunking.

3 Implementation in the Futhark Compiler

Futhark is a purely functional data-parallel array language that supports regular nested parallelism, with a compiler capable of generating efficient OpenCL code for GPUs. We have modified the Futhark compiler to handle segmented reductions via the three different strategies presented previously. Since the topic of this paper is not Futhark itself, we shall go into little detail on the language as such, and only describe concepts and (non-obvious) syntax as necessary.

Syntactically, Futhark resembles a combination of OCaml and Haskell. Parallelism is expressed via built-in *second-order array combinators* (SOACs), such as **map** and **reduce**. A segmented reduction is not a SOAC by itself, but occurs naturally when a **reduce** is nested inside of a **map**. For example, the function `sumrows` shown on Figure 7a sums the rows of an array `xss`.

The function `sumrows` takes a single parameter, `xss`, which is a two-dimensional $m \times n$ array of 32-bit integers, written as the type `[m][n]i32`. The `[m]` `[n]` annotations prior to the parameter are implicit size parameters, and indicate that the function is polymorphic

```
1  let sumrows [m] [n] (xss: [m][n]i32): [n]i32 =
2    map (λxs → reduce (+) 0 xs) xss
```

(a) Summing the rows of a matrix of integers in Futhark.

```
1  let sumrows [m] [n] (xss: [m][n]i32): [n]i32 =
2    map (λxs → reduce (+) 0 (map i32.abs xs)) xss
```

(b) Summing the absolute value of the rows of a matrix of integers in Futhark.

Figure 7. Two simple examples of segmented reductions in Futhark.

```
1  let main [m] [n] (xss: [m] [n]i32) =
2    map (λxs → let v = reduce (+) 0 xs
3               in reduce (*) 1 (map (+v) xs)) xss
```

(a) A **map** containing two **reduce** operations, which cannot by itself be turned into a segmented reduction, as the nesting is imperfect.

```
1  let main [m] [n] (xss: [m] [n]i32) =
2    let vs = map (λxs → reduce (+) 0 xs) xss
3    in map (λ(v,xs) → reduce (*) 1 (map (+v) xs)) vs xss
```

(b) The **map** operation has been split into two by fission, each of which contains a perfectly nested **reduce**.

Figure 8. Transforming imperfectly nested reductions to perfectly nested reductions, so they can be translated to segmented reductions.

in the two sizes m and n (as opposed to the size of the xss array being determined by two variables n and m in scope).

A common pattern is to reduce the result of a **map**, such as computing the sum of the result of applying the function **i32**.abs to all elements of xs, as on Figure 7b. This pattern is recognized by the Futhark compiler and fused into a special internal construct, called **redomap**[14], that will effectively compute both the reduction and the application of **i32**.abs in one pass over the data in xs . This can be a significant improvement as both **map**s and reductions are typically memory bound computations. Just as with reductions, we often see the pattern of a *segmented redomap*. In this paper we will not distinguish between the two, and use exclusively the term "segmented reduction", even for programs that involve fusion between **map** and **reduce**.

Finally, we note that while this paper is concerned exclusively with perfectly nested segmented reductions, where a **reduce** is the only component of a function being mapped (possibly after fusion with a **map**), real programs may exhibit more complicated nesting structures. For example, 8a shows a program that applies a function containing two **reduce** operations on each row of a matrix. By itself, this does not correspond to a segmented reduction, but the Futhark compiler is able to automatically transforms the program into the form shown on 8b, which now corresponds to two perfectly nested segmented reductions. This technique has been published elsewhere [15] and is not the subject of this paper. The important point is that not having segmented reductions as a language primitive is not a great hindrance, as the compiler can automatically restructure a program to contain perfectly nested reductions.

3.1 Choosing a Strategy

When the Futhark compiler encounters such a nested reduction (possibly after automatic transformation), it will generate code for each of the three strategies for segmented reduction, with branches inserted to select a strategy at runtime. The present heuristic are:

1. If more than 2^{16} segments are present, use the sequential segments strategy.
2. Otherwise, if the segment size is greater than half the workgroup size, use the large segments strategy.
3. Otherwise, use the small segments strategy.

The rationale is that we prefer to sequentialize the reduction whenever there is enough work to fully occupy the hardware, because a parallel reduction requires inter-thread communication/synchronization overhead. Failing that, we prefer the large-segment strategy because it essentially translates to workgroup level reduction – a simpler operation with a known efficient implementation – and allows efficient chunking (multiple memory transaction in flight). For the rest of the cases we use small-segment reduction, which still allows an efficient implementation based on workgroup-level segmented scan (that uses fast memory and workgroup-level synchronization).

As we shall see in Section 3.2 this heuristic are not optimal for all programs. The optimal choice is dependent on both the concrete GPU hardware on which we execute, as well as the characteristics of the reduction operator in use. However, the heuristic above tends to perform acceptably well in practice. Improving the procedure for selecting a strategy remains future work.

3.2 Microbenchmarks

We have implemented four different benchmarks to demonstrate the performance characteristics of the implementation of segmented reduction in the Futhark compiler. The benchmarks are executed on an NVIDIA K40 GPU. We use 512 workgroups, and a workgroup size of of 256 threads.

For each benchmark, we measure runtime for each of the three kernel strategies (large segments, small segments, and sequential segments). We use two work sizes: a *small* one comprising 2^{18} elements, and a *large* one comprising 2^{26}. As before, we vary the ratio between number of segments and segment size, while keeping the work constant.

As frames of reference, we measure the runtime of an implementation that uses a segmented scan to implement segmented reduction, as well as a *non-segmented* reduction using the same operator. This latter reference does not compute the same result, but shows to which degree the segmented operation (which is conceptually more complicated) is slower than a corresponding non-segmented one.

We are primarily demonstrating how the various strategies perform on various workloads, and only secondarily concerned with absolute runtimes. However, to show performance compared to prior (non-Futhark) solutions, we have also implemented two of the micro-benchmarks in CUDA C++ with the help of NVIDIAs CUB library (version 1.7.0), which supports (potentially irregular) segmented reductions via the `cub::DeviceSegmentedReduce` class. CUB is well regarded for its performance, due to careful attention paid to issues of hardware-specific low-level optimisation. In particular, CUB uses more efficient intra-group communication than the code generated by Futhark.

For segmented reductions, CUB's strategy is to launch a workgroup for each segment, which then reduces the segment. As we shall see, while CUB outperforms Futhark for inputs that hit the sweet spot, CUB performs poorly for edge cases with many small or a few large segments, and fails entirely when the number of segments exceeds the maximum number of workgroups supported by the GPU (2^{16}). While this latter problem could easily be remedied, performance would be poor for inputs that contain a large number of small segments, as most of the threads in each workgroup would be idle. CUB computes its desired workgroup count and workgroup size internally.

Unfortunately, the machine on which the benchmarks have been performed exhibits significantly higher GPU kernel launch latency for OpenCL (the API used by Futhark) than for CUDA (the API used by CUB) on some kernels, on the order of 400μs. On small workloads, this launch latency becomes dominant, resulting in CUB outperforming Futhark to a greater degree than can be attributed to its efficient low-level optimisations. On larger workloads, this effect has no significant impact on the results.

We note that Futhark's implementation of non-segmented reduction has been shown previously to match or exceed the performance of the implementation in Thrust [14]. Apart from benchmarking each strategy by itself, we also measure the runtime of *automatically* picking a strategy, which is the compiler inserting code for all three strategies, and determining which one to activate at runtime based on the heuristic mentioned in Section 3.1.

The benchmarking infrastructure and all implementations are publicly available at

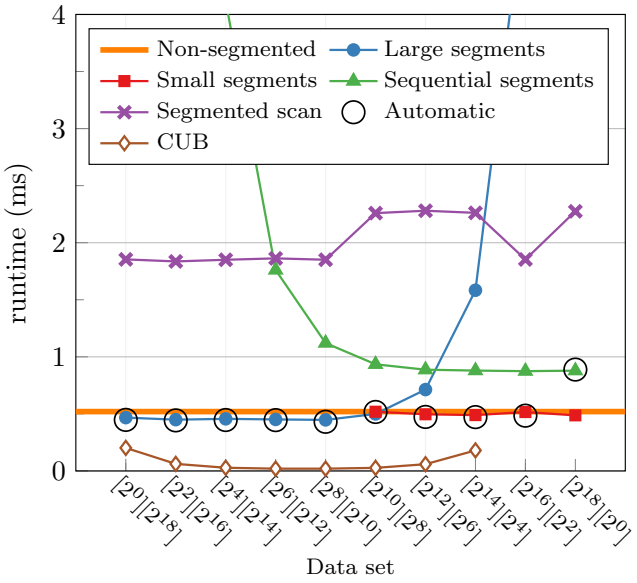
https://github.com/HIPERFIT/futhark-fhpc17

Segmented Sum (implementation on Figure 7a and runtimes on Figure 9) is a straightforward computation of the sums of rows of a matrix, as shown previously. Note on Figure 9a the switch from small segments to sequential segments when the number of segments reaches 2^{16}, despite the small segments strategy still being more efficient at this point. The reason for the sequential segments strategy being slower is the cost of a transposition inserted to ensure coalesced memory access.

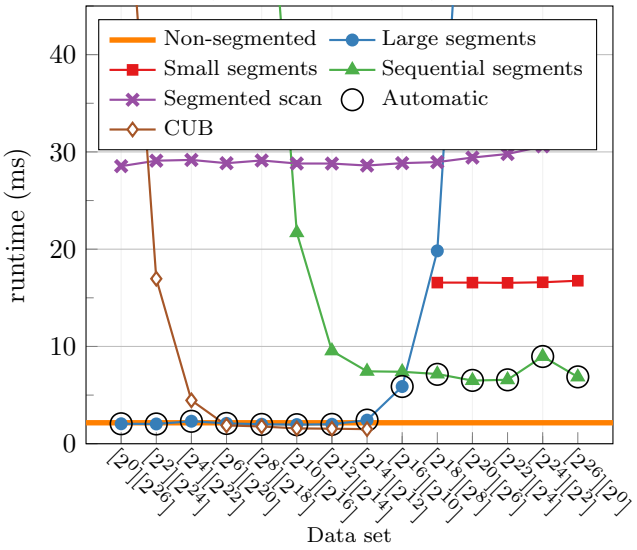
Index of Max (implementation on Figure 10 and runtimes on Figure 11) computes the largest element of each row of an array. The amount of global memory traffic is not much more than for segmented summation, as the `iota` (which produces an array from 0 to n) is fused by the Futhark compiler. However, twice as much local memory is needed, as both the value and its corresponding index is tracked.

While the Futhark compiler can recognise the commutativity of certain simple functions (like addition), it cannot recognise that the function used here is commutative. Hence, we use reduce_comm instead of just **reduce** as a hint to the compiler.

One complication is the need to provide a neutral element for the reduction. In general, for the function we are using, there is no fully neutral element. However, using a pair of the smallest 32-bit integer at one position past the end of the array suffices, as this will never be chosen over any element of the array. Only if the input array is empty will this "neutral element" be produced by the reduction.



(a) Segmented sum for 2^{18} integers.



(b) Segmented sum for 2^{26} integers.

Figure 9. Runtime measurements for segmented summation. The implementation is shown on Figure 7a.

The observed performance characteristics are similar to Segmented Sum, except that the small segments strategy deteriorates to have approximately the same performance as sequential segments when reaching 2^{18} segments on the small dataset.

Maximum Subarray Sum (implementation on Figure 12 and runtimes on Figure 13) finds, within each row, the contiguous subarray which has the largest sum.[5] This is expressed as a composition of **map** and **reduce**.

[5]This problem is often called the *Maximum Segment Sum*, but this term might be confusing given our other uses of the term "segment".

```
1  let redop ((xv, xi): (i32,i32)) ((yv, yi): (i32,i32)): (i32,i32) =
2    if xv < yv then (yv,yi)
3    else if yv < xv then (xv,xi)
4    else -- Prefer lowest index if the values are equal.
5    if xi < yi then (xv,xi) else (yv,yi)
6
7  let index_of_max [n] (xs: [n]i32): i32 =
8    let ne = (-2**31-1, n)
9    let (_, i) = reduce_comm redop ne (zip xs (iota n))
10   in i
11
12 let main [m] [n] (xss : [m][n]i32) : [m]i32 =
13   map index_of_max xss
```

Figure 10. Finding the index of the largest element in each of the segments.

Unlike the other benchmarks, the reduction operator for Maximum Subarray Sum is not commutative (although it is of course still associative). This means that each segment must be reduced in its original order. Unfortunately, having each GPU thread iterate sequentially through memory leads to inefficient *non-coalesced* memory accesses. The Futhark compiler automatically detects and resolves this issue via an index-space transformation of the original input array, followed by changing the iteration order of the threads. The full details of this technique are the subject of a previous publication [14].

No CUB implementation exists, because CUB requires the reduction operator to be commutative, which the MSS operator is not.

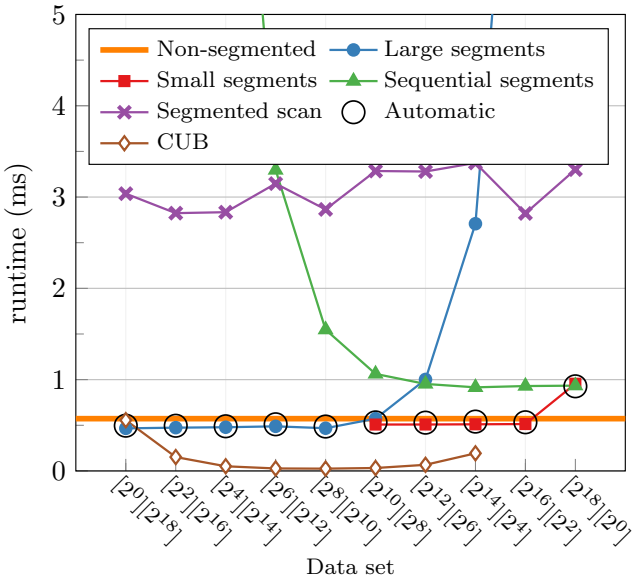
Black-Scholes Option Pricing (implementation on Figure 14 and runtimes on Figure 15) employs a composition of **map** and **reduce** to compute the value of m options running at n days each. Each segment corresponds to an option with a unique volatility and risk-free rate. The reduction operator is a trivial addition, but the function used for the **map** part is complicated.

Unlike the other benchmarks, the segments are not part of the input, but are constructed with **iota**, which is then fused into the inner reduction. As a result, this benchmark exhibits much less memory traffic than the others, and its performance is in fact bound by the computational capacity of the GPU, not by the memory bandwidth. Yet, we see the same performance characteristics as for the memory-bound benchmarks.

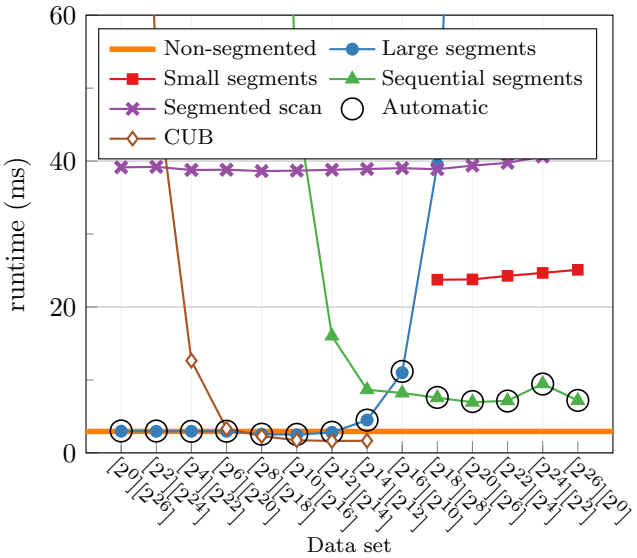
The CUB implementation has been written with a custom iterator that performs the equivalent of the mapop operation on demand, in order to avoid the need for an intermediate array.

4 Impact on Two Rodinia Benchmarks

This section shows the impact of replacing the Futhark compiler's previous implementation of regular segmented reductions, which was based on scans, with the new implementation described in this paper. This is done by measuring the performance of two benchmarks taken from the well-known Rodinia benchmark suite:



(a) Index of largest element for 2^{18} integers.



(b) Index of largest element for 2^{26} integers.

Figure 11. Runtime measurements for finding the index of the largest element in each segment. The implementation is shown on on Figure 10.

Backprop and K-means. Both of these contain regular segmented summations. We use the OpenCL implementations from version 3.1 of Rodinia.

Figure 16 shows for each benchmark the runtime of the hand-written OpenCL reference implementation from Rodinia, the runtime of a Futhark implementation where segmented reductions are implemented using scans, and the runtime of a Futhark implementation where segmented reductions are implemented by automatically picking between the strategies presented in Section 2. We use Futhark's default configuration for workgroup size (256 threads) and number of workgroups (128).

```
1  type quad = (i32,i32,i32,i32)
2
3  let redop ((bx, lx, rx, tx): quad) ((by, ly, ry, ty): quad): quad =
4    (i32.max bx (i32.max by (rx + ly)),
5     i32.max lx (tx+ly),
6     i32.max ry (rx+ty),
7     tx + ty)
8
9  let mapop (x: i32): quad =
10   (i32.max x 0, i32.max x 0, i32.max x 0, x)
11
12 let mss [n] (xs: [n]i32): i32 =
13   let (x, _, _, _) = reduce redop (0,0,0,0) (map mss.mapop xs)
14   in x
15
16 let main [m] [n] (xss: [m][n]i32) =
17   map mss xss
```

Figure 12. Implementation of segmented maximum subarray sum.

In Backprop, the segmented summation is straightforward and appears as a perfectly nested reduction in the source code. Previously, executing the segmented summation took up 26% of the total runtime, which has been cut to approximately 2% with the new implementation. For the dataset used, the segmented reduction involves 16 segments each containing 10^{20} elements, corresponding to 8 workgroups per segment, and each thread processing 512 elements sequentially.

For K-means, two different datasets are used. Both perform K-means clustering on a 35-dimensional space, with the `204800` dataset dividing 204,800 points into eight clusters, and `kdd_cup` dividing 494,019 points into five clusters. In contrast to the straightforward segmented summation in Backprop, it takes a bit of work for the compiler to recognise it in K-means. One case arises from computing the number of points in each cluster, in which a subcomputation involves a reduction over a matrix, where the reduction operator is vector addition[6]. In Futhark, this is written as

reduce (λx y $\rightarrow$ **map** (+) x y) (**replicate** k ne) arr

where arr has shape [n][k]. Since a reduction with an array-typed operand type can be very expensive, the compiler automatically interchanges the **reduce** and the **map** (at the cost of a transposition), producing

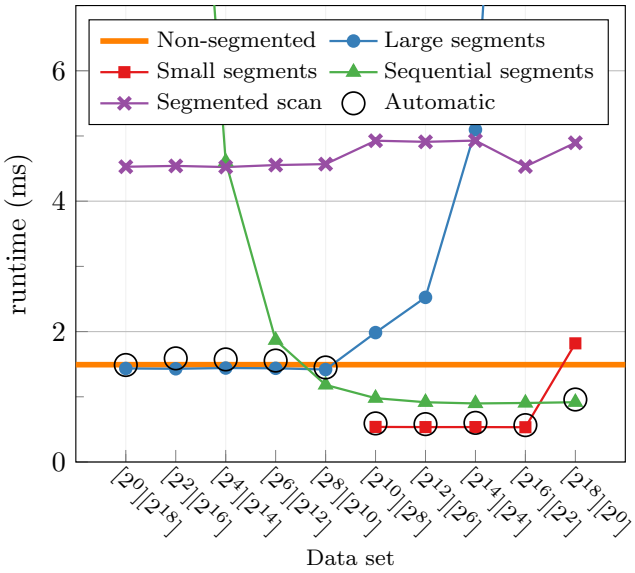
map (λrow $\rightarrow$ **reduce** (+) ne row) (**transpose** arr)

which is exactly a segmented reduction over k segments, each of size n. For our two datasets, k is small (five or eight), while n is large (494,019 or 204,800). The large-segments strategy is best here.

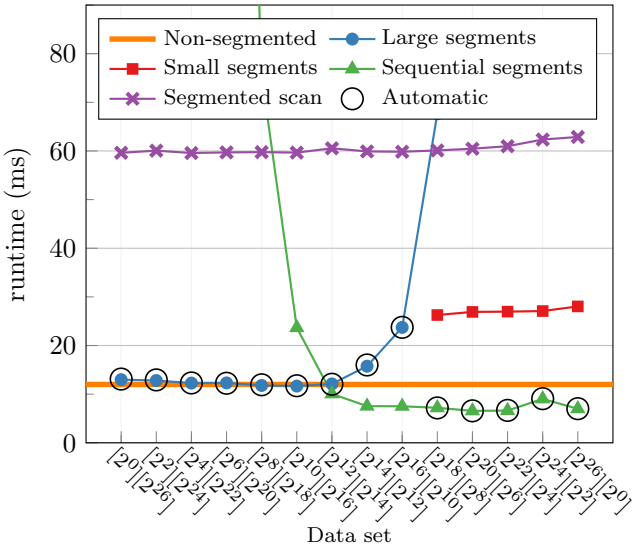
Another segmented reduction occurs when computing cluster centers. Here the reduction operator is matrix addition, and the result contains two nested **map**s outside of a **reduce**.

For `kdd_cup`, the proportion of runtime spent on computing segmented reductions declines from 19% to 1%, and for `204800` from 35% to 2%. In the latter case, it is the difference between Futhark being 1.35× slower than the reference implementation, and being 1.29× faster.

[6]The code is slightly more elaborate than a plain **reduce**, as it involves **stream_red**, a special Futhark construct that permits efficient sequentialisation, from which the **reduce** is extracted as a secondary stage



(a) Maximum subarray sum for 2^{18} integers.



(b) Maximum subarray sum for 2^{26} integers.

Figure 13. Runtime measurements for determining the maximum subarray sum of each segment. The implementation is shown on Figure 12.

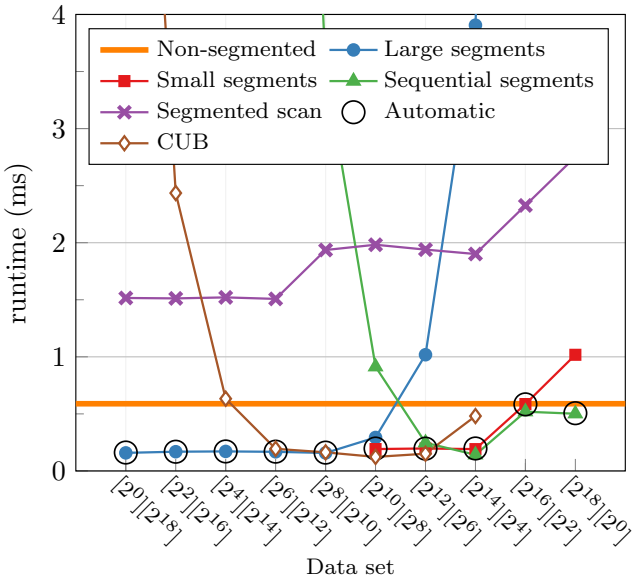
5 Related Work

In a broad sense, a segmented reduction is a special case of a histogram computation, where an input array of key-value pairs are processed, to reduce all the values with the same key. This is illustrated in imperative pseudo-code on Figure 17.

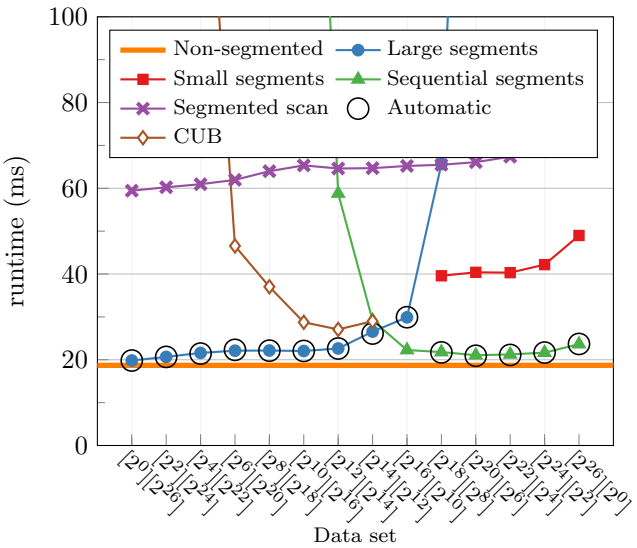
Some implementations of segmented reduction use a definition similar to the histogram computation, with the restriction that a segment is defined as a consecutive range of matching keys (i.e., the keys 1 1 2 1 defines 3 segments). Because there is no restriction on the size of a segment, this is called an *irregular segmented reduction*. While there has been little prior work on regular segmented reductions, irregular reductions have received some attention.

```
1  let horner (x: f64): f64 =
2    let (c1,c2,c3,c4,c5) =
3      (0.31938153,-0.356563782,1.781477937,
4       -1.821255978,1.330274429)
5    in x * (c1 + x * (c2 + x * (c3 + x * (c4 + x * c5))))
6
7  let cnd0 (d: f64): f64 =
8    let k = 1.0 / (1.0 + 0.2316419 * f64.abs d)
9    let p = horner k
10   let rsqrt2pi = 0.39894228040143267793994605993438
11   in rsqrt2pi * f64.exp(-0.5*d*d) * p
12
13 let cnd (d: f64): f64 =
14   let c = cnd0 d
15   in if 0.0 < d then 1.0 - c else c
16
17 let mapop (r: f64) (v: f64) (days: i32) (day: i32): f64 =
18   let call = day % 2 == 0
19   let price = 58.0 + 4.0 * f64 (1+day) / f64 days
20   let strike = 65.0
21   let years = f64 (1+day) / 365.0
22   let v_sqrtT = v * f64.sqrt years
23   let d1 =
24     (f64.log(price / strike) + (r + 0.5 * v * v) * years) /
25     v_sqrtT
26   let d2 = d1 - v_sqrtT
27   let cndD1 = cnd d1
28   let cndD2 = cnd d2
29   let x_expRT = strike * f64.exp (-r * years)
30   in if call then price * cndD1 - x_expRT * cndD2
31     else x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1)
32
33 let blackscholes (r: f64) (v: f64) (n: i32): f64 =
34   reduce (+) 0.0 (map (mapop r v n) (iota n))
35
36 let main [m] (rs: [m]f64) (vs: [m]f64) (n: i32): [m]f64 =
37   map (λr v → blackscholes r v n) rs vs
```

Figure 14. Implementation of Black-Scholes option pricing, with each segment corresponding to a unique value for r and σ, and the size of the segment the running time of the option.



(a) Black-Scholes pricing for 2^{18} days in total.



(b) Black-Scholes pricing for 2^{26} days in total.

Figure 15. Runtime measurements for segmented Black-Scholes pricing. The implementation is shown on Figure 14.

The work on the data parallel programming language NESL [5], showed how a segmented reduction, capable of handling irregular segments, could be implemented by using a segmented scan, and an instruction to get the last element from each segment. The only requirement is that the reduction operator is associative. NESL also shows that a segmented scan can be implemented efficiently by using scans and a flag array [4]. While this approach exploits all available parallelism and has performance mostly invariant of the segment size, our results from Section 3.2 show that it is slower than specialised implementations of segmented reductions.

The Modern GPU [1] library from NVIDIA supports irregular segmented reductions, and supports using either using keys or offsets as the segment descriptor. Irregular segments are handled by assigning an equally sized slice of the input array to each workgroup, where each thread will process multiple consecutive elements. Partial results for a segment are combined using a special streaming kernel that scans carry-out values and redistributes them into the destination reductions.

In essence, these approaches solve a more general problem, of which regular-segmented reduction is just an instance of, and consequently result in higher overheads.

A more related strand of research are DSLs aimed at parallel programming such as Accelerate [6] and Delite [17], which provide specialized implementations for regular segmented reduction. Accelerate always process one segment in one GPU workgroup, while Delite maps inner reductions (parallelism) at warp or workgroup level. These approaches are inefficient in some cases: for example if the array consists of only few large segments, then hardware

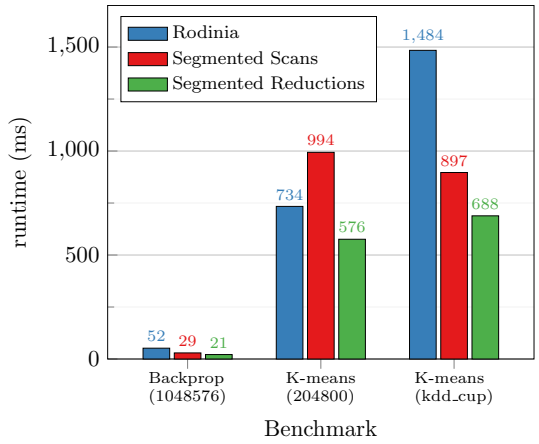


Figure 16. The performance impact of using the segmented reduction implementation presented in the paper over a previous implementation based on segmented scans.

```
1  -- 'input_key' and 'input_val' are arrays of size 'n'
2  -- 'res' is a dictionary-like datastructure
3  for i in [0...n]:
4    acc = res[ input_key[i] ]
5    res[ input_key[i] ] = OP(acc, input_values[i])
```

Figure 17. A histogram computation in imperative pseudocode.

parallelism would be severely underutilized. We saw this in the results for CUB in Section 3.2, which uses a similar approach.

In comparison, our approach automatically adapts to the dataset, by choosing at runtime one of the three evaluation strategies, which cover efficiently all cases of number of segments and segment size.

6 Conclusions and Future Work

This paper has shown three different implementation strategies for regular segmented reduction on GPUs, each optimised for a specific range of segment sizes and number of segments. We have demonstrated a prototype of a compiler that can dynamically switch between the strategies based on the workload encountered at runtime. The performance of the generated code exceeds that which is achieved with a scan-based approach, without exhibiting any significant slowdown for extreme cases with very few large segments or many very small segments.

We have shown that regular segmented reductions occur as subcomputations in real published benchmark suites, and that their efficient implementation has a non-negligible impact on overall runtime.

Work remains on improving performance for segmented reductions with non-commutative operators, as well as refining the decision procedure used for choosing a reduction strategy for a given dataset. The latter must likely be solved using autotuning techniques.

Acknowledgments

We are grateful to NVIDIA for donating the K40 GPU used for this work.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (http://hiperfit.dk) under contract number 10-092299.

References

[1] S. Baxter. Modern GPU 1.0, 2013. URL https://moderngpu.github.io/segreduce.html.

[2] L. Bergstrom and J. Reppy. Nested data-parallelism on the Gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 247–258, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364563. URL http://doi.acm.org/10.1145/2364527.2364563.

[3] R. Bernecky and S.-B. Scholz. Abstract expressionism for parallel performance. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 54–59, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3584-3. doi: 10.1145/2774959.2774962. URL http://doi.acm.org/10.1145/2774959.2774962.

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

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

[6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Procs. of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Procs. of IEEE Int. Symp. on Workload Characterization (IISWC)*, pages 44–54, Oct 2009. doi: 10.1109/IISWC.2009.5306797.

[8] M. Dybdal, M. Elsman, B. J. Svensson, and M. Sheeran. Low-level functional GPU programming for parallel algorithms. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 31–37. ACM, 2016.

[9] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*, pages 15–24. ACM, 2011.

[10] M. Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[11] T. Henriksen and C. E. Oancea. Bounds Checking: An Instance of Hybrid Analysis. In *Procs. of ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627388. URL http://doi.acm.org/10.1145/2627373.2627388.

[12] T. Henriksen, M. Elsman, and C. E. Oancea. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *Procs. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC'14, pages 31–42, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3040-4. doi: 10.1145/2636228.2636238. URL http://doi.acm.org/10.1145/2636228.2636238.

[13] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, FHPC'16, pages 38–43, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4433-3. doi: 10.1145/2975991.2975997. URL http://doi.acm.org/10.1145/2975991.2975997.

[14] T. Henriksen, K. F. Larsen, and C. E. Oancea. Design and GPGPU Performance of Futhark's Redomap Construct. In *Procs. of the 3rd ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'16, pages 17–24, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4384-8. doi: 10.1145/2935323.2935326. URL http://doi.acm.org/10.1145/2935323.2935326.

[15] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. 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 '17, New York, NY, USA, 2017. ACM.

[16] J. Hoberock and N. Bell. Thrust: A parallel template library, 2016. URL http://thrust.github.io/.

[17] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Procs. of the 47th Annual IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-47, pages 63–74, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.23.

[18] D. Merrill. CUB. https://github.com/NVlabs/cub, 2017.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.