Generating Work Efficient Scan Implementations for GPUs the Functional Way

Federico Pizzuti, Michel Steuwer, Christophe Dubach

[1] University of Edinburgh
[2] McGill University

Abstract. Scan is a core parallel primitive. High-performance work-efficient implementations are usually hard-coded, leading to performance portability issues. Performance portability is usually achieved using a generative approach, which decomposes the primitive in simpler, composable parts, expressing the implementation space.
Data parallel functional languages excel at expressing programs as composition of simple patterns. Lift, Furthark, Accelerate have successfully applied this technique to patterns such as parallel reduction and tiling. However, work-efficient parallel scan is *still* provided as a hard-coded builtin.
This paper shows how to decompose a classical GPU work-efficient parallel scan in terms of other data-parallel functional primitives. This enables the automatic exploration of the implementation design space, using a set of simple rewrite rules.
As the evaluation shows, this technique outperforms hand-written baselines and Furthark, a state of the art high performance code generator. In particular, this composable approach achieves a speedup of up to 1.5× over hard-coded implementations on two different Nvidia GPUs.

1 Introduction

Parallel hardware offers great opportunities for performance but is difficult to program. Modern parallel architectures are complex and it is hard for developers to fully exploit their potential. A compiler-oriented approach is highly desirable to exploit these systems automatically.

Data Parallel Functional Code Generators have been shown to be a viable solution in tackling this challenge. Projects such as Lift[20], Futhark[8], and Accelerate[3], are capable of taking a high-level program and automatically generating high-performance implementations, targeting a variety of platforms. Functional representations possess a number of desirable characteristics: the lack of side effects allows one to easily reason about parallelism, the emphasis on function composition maps well to the idea of parallel patterns, and finally the rich type systems allow the expressions of powerful invariants, enabling very sophisticated program optimizations.

The approach is not without downsides. Many algorithms and applications which are relatively straightforward to implement in a traditional low-level language are not as clearly implemented in a functional style. This includes common

This preprint has not undergone any post-submission improvements or corrections.
The Version of Record of this contribution is published in “Euro-Par 2022: Parallel Processing”, and is available online at https://doi.org/10.1007/978-3-031-12597-3

application domains and algorithms for which a plethora of well-known implementations are readily available. This gap is due both to the highly constrained nature of functional languages and their relative obscurity in the realm of high-performance code generation for parallel hardware.

The most salient example of such a missing algorithm is *scan*, a core data-parallel control pattern [11]. *Scan* is a crucial component to many application domains, such as linear algebra [14] and computer graphics. There is extensive literature both concerning the algorithmic approaches to the parallelization of scan [4] and the concrete techniques to be used on specific platforms [19, 12].

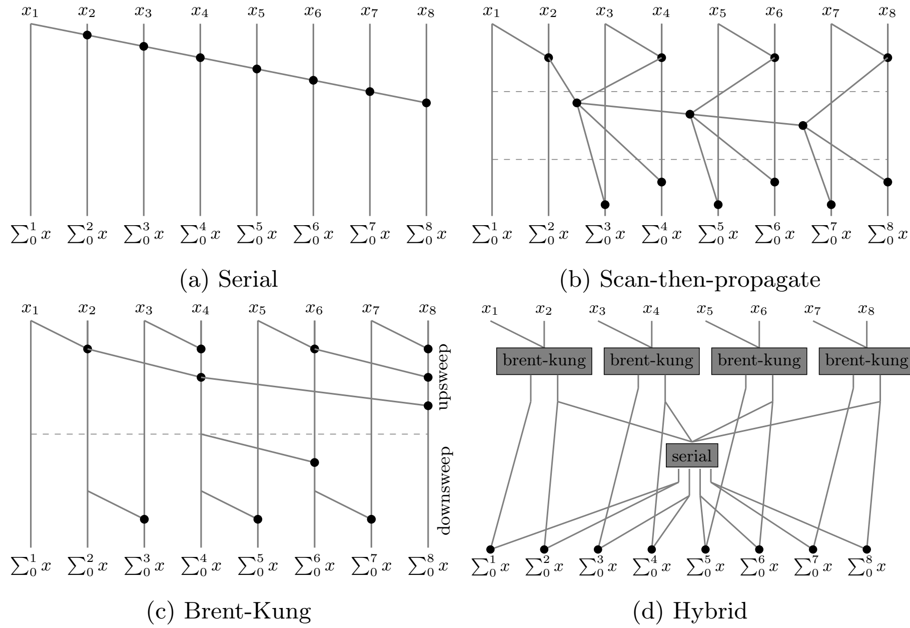
However, in most data-parallel functional code generators, *scan* is either only available as a sequential primitive, such as in the case of Lift compiler, or as a parallel primitive with a black-box implementation, such as in the case of Futhark. While the latter approach offers at least some way to express a program using a *parallel scan*, it requires the compiler authors to provide a handwritten implementation, either precluding the usage of the compiler's own powerful optimization techniques or applying them in an ad-hoc way.

This paper addresses these shortcomings by deriving a functional formulation of *parallel scan*, expressed within a data-parallel functional programming language. As the compiler used has OpenCL as its primary target, the implementation used is optimized for a GPU system. The technique used is based on publicly available code by NVidia [7].

The papers then demonstrates how to decompose a *parallel scan* implementation into a number of *reusable* and *composable* rewrite rules, modeling the algorithm's *optimization space*. These rewrites can be used to optimize arbitrary *scan* calls, rewriting them into a parallel implementation. As the rules model an optimization space, the compiler can automatically explore possible variations, leading to significant performance gains across different GPU architectures. - The automatic derivation of a parallel implementation of scan is a thoroughly studied topic. A generic technique for deriving a scan parallelisation is given in[13]. However, such schemes do not generate *work-efficient* implementations. A parallel algorithm is said to be work efficient if its asymptotic complexity is at most a constant factor away from the best known sequential implementation. To the best of the authors' knowledge, the method presented here is the first to yield a *work-efficient* parallel scan from a high-level specification and suitable for a GPU target.

In summary, this paper makes the following contributions

- presents a functional formulation for a GPU *parallel scan*, based on a number of parallelization strategies.
- decomposes the functional formulation into a number of rewrite rules, modeling a parametric space of parallel scan algorithms that can be mechanically explored to derive efficient GPU scan implementations.
- evaluates the overall effectiveness of the approach, by comparing the generated code it with a handwritten reference implementation and the state-of-the-art parallel code generator Futhark, outpeforming both by a factor of $1.5\times$



(a) Serial

(b) Scan-then-propagate

(c) Brent-Kung

(d) Hybrid

Fig. 1: Scan

2 Background

2.1 Parallel Scan

Scan is one of the fundamental data parallel control patterns [11]. The semantics of scan are illustrated in the following equation:

$$scan(\oplus, x_0, [x_1, \dots x_n]) = [x_0, x_0 \oplus x_1, \dots, x_0 \oplus \dots \oplus x_{n-1}]$$

Given an associative binary operator a starting value and an input sequence, generates a new sequence of n elements in which the i-th item is the result of recursively applying the operator i times.

Being a well-studied operation, there is a wide number of known scan implementations[9], with varying complexity and parallelism. Relevant strategies for this paper include *Serial* scan(figure 1a), *Scan-then-propagate*(figure 1b) and the *Brent-Kung* scan(figure 1c). The latter is both parallel and *work-efficient*.

Additional scan implementations may be derived by combining together different strategies. This is common when implementing parallel scan for devices such as GPUs, and is the approach used by a well-known publicly available NVidia implementation [7]. The algorithm, shown as (figure 1d) unfolds on two levels: an outer scan following the scan-then-propagate strategy, and an inner scan implementation parallelized using the *Brent-Kung* approach.

$\boldsymbol{map} : (T \to U) \to [T]_N \to [U]_N$

$\boldsymbol{split} : N \to [T]_{N \cdot M} \to [[T]_N]_M$

$\boldsymbol{join} : [[T]_N]_M \to [T]_{N \cdot M}$

$\boldsymbol{reduce} : (T \to T \to T) \to T \to [T]_N \to T$

$\boldsymbol{scan} : (T \to T \to T) \to T \to [T]_N \to [T]_N$

$\boldsymbol{zip} : [T]_N \to [U]_N \to [(T, U)]_N$

Fig. 2: Data parallel patterns and their types.

2.2 Data Parallel Functional Code Generators

Data Parallel Functional Code Generators are compilers that take as input a programs written in a functional programming language or expressed as a functional IR to generate high-performance code targeting parallel architectures. The functional style is used to express parallel programs as compositions of *primitive* functions. These primitives often encode basic data-parallel patterns, such as *map* and *reduce*, data reordering transformations such as *split* and *join*, and elementary scalar operations.

This work has been implemented in a dialect of Lift, a data parallel language used in a wide range of applications[20, 21, 6, 14].

The preferred data structure is the array, nested or multidimensional at the language level, but often represented as a flat contiguous buffer in the generated code. As the language is purely functional, arrays are never mutated. Rather, patterns always produce new arrays. It is important to note that most of these transformations are lazy whenever possible, avoiding spurious copies.

Data parallel functional languages tend to have rich type systems. This enables the compiler to statically check invariants that otherwise either go unchecked, or are implemented in terms of dynamic checks at run-time. For example, the length of each arrays is tracked at the type level as a symbolic algebraic expression. In this paper, every array has it's length represented as a symbolic formula. Examples of this can be seen in the patterns described in figure 2.

3 Functional Formulation of Work Efficient Parallel Scan

This section presents a work-efficient GPU implementation of parallel scan expressed in the Lift dialect mentioned in the previous section. The code is an adaptation of an handwritten NVidia implementation, whose imperative pseudocode is also shown. Algorithmically, this is a case of hybrid scan, as shown in figure 1d, and can be analyized in terms of the *outer* and *inner* scans.

3.1 Outer Scan

Listing 1 shows the functional formulation of the outer scan side-by-side with the equivalent imperative pseudo-code. The algorithm unfolds in three sections: the block-scan phase, the global scan phase, and then the aggregation phase.

In the block scan phase, the data input is split in blocks of size `BLK`, and each `block_scan` is computed in parallel. `unzip` is then used to separate an array of

```
1 parallel for i=0 to BLK do
2  block_scan(data[i*N/BLK], &sums[i])
3  sequential_scan(sums, global_scan)
4  parallel for b=0 to BLK do
5  parallel for i=0 to N/BLK do
6   data[b*BLK+i] = data[b*BLK+i] + global_scan[b]
```

```
1 let (data',sums) = data|> split(BLK)|> map(block_scan)|> unzip
2 let global_scan = sums|> scanSeq(+)
3  zip(data', global_scan)|> map(
4    (xs, b) => xs|> map(x =>x+b)
5  )|> join
```

Listing 1: Imperative pseudo-code and functional expression for the outer scan.

`[(partial_scan, sum)]` pairs into a pair of `([partial_scan],[sum])` arrays. This corresponds to the multiple output parameters in the imperative pseudo-code. `global_scan` is implemented sequentially, by calling the `scanSeq` primitive.

Finally, we reach at the aggregation step. The `zip` primitive associates each block's partial scan with the corresponding overall `global_scan`. The outer `map` call operates over these `(block, value)` pairs, adding `value` to every element of `block`. The blocks are concatenated by using `join`, which flattens the array.

It must be remarked that the functional version diverges somewhat from the imperative version, which freely updates arrays in-place, something which is difficult to express in a language that forbids. This has forced the introduction of some intermediate variables, such as `data'` in listing 1.

3.2 Inner Scan

The inner `block_scan` is more complex to express faithfully in a functional style. Just as in the case of the outer algorithm, we will have to give up on an in-place formulation. However, the inner algorithm performs in-place writes within sequential for loops, which prevents the introduction of intermediate variables.

Listing 2 demonstrates how to remedy the issue in the *upsweep* phase. Notice that the sequential iterations are in fact not dependent on the size of the input but rather the fixed parameter `BLK`: implying the loop can be unrolled. Afterward, we can proceed by introducing the intermediate variables in place of every mutation step. This implies that the *upsweep* phase therefore no longer updates the tree in place. Rather it constructs the tree level by level, storing each successive layer in a different variable.

```
1  for d=0 to log2(n-1) do
2   parallel for k=0 to n-1 by 2^(d+1) do
3    x[k+2d] = x[k+2d-1]+x[j+2d]
```

```
1  // d = 0...
2  let up_0 = input |> split(2) |> map(+) |> join
3  // d = 1...
4  let up_1 = up_0 |> split(2) |> map(+) |> join
5  //....
6  let up_n = up_n-1 |> split(2) |> map(+) |> join
```

Listing 2: Imperative pseudo-code and functional expression of *upsweep* phase

With this knowledge, we notice that each iteration produces the next tree layer by summing together adjacent pairs of the previous tree layer, expressed functionally by chaining together the primitives `split(2)|> map(+)|> join`.

Likewise, listing 3 shows how the *downsweep* phase is unrolled to combine the generated layers in Last-In-First-Out order. We begin the recursion with an array containing a single element, 0. This corresponds to the line `x[n-1]=0` in the imperative code. The layers are then combined, via executing `zip(prev_layer, layer |> split(2))|> map(scan(+, 0))|> join`, which is in fact equivalent to the seemingly-unrelated loop body in the imperative pseudo-code: given two layers, we match together one element of the previous (smaller) layer with two elements of the successive, larger layer, with the pseudo-code lines 10-12 being an in-lined sequential scan. The importance of generalizing this custom-looking code into a *scan* call is shown in section 4.2. Finally, the functional version must return the last tree layer and the block sum value, as the tuple `(down_0, up_n [0])`.

We have now derived a working functional formulation of the GPU work efficient scan, and have encountered some limitations of the functional approach, such as introducing intermediate variables have been introduced for in-place updates – although this limitation can be overcome in certain cases, such as when the intermediate layers are used only once (such as the case for `up_n` and `down_n`. As we will see in the section 6, even when fusion is not possible, these intermediates do not significantly affect the performance of the generated code.

4 Modeling the Optimization Space with Rewrite Rules

The aim of this section is to derive an optimization process capable of mapping uses of the *scan* pattern to work-efficient parallel implementations. We will do so by generalizing the functional formulation presented in section 4.

```
5  sum = x[n-1]
6  x[n-1] = 0
7  for d = log2(n-1) down to 0 do
8   parallel for k = 0 to n-1 by 2^(d+1) do
9    t = x[k + 2d - 1];
10   [k + 2d - 1] = x[k + 2d]
11   x[k + 2d] = t + x[k + 2d]
```

```
7  let down_n = [0]
8  let down_n-1 =
9   zip(down_n, up_n |> split(2)) |>
10  map(scanSeq(+)) |> join
11 ....
12 let down_0 =
13   zip(input, up_0 |> split(2)) |>
14   map(scanSeq(+)) |> join
15 (down_0, up_n[0])
```

Listing 3: Imperative pseudo-code and matching functional expression of *down-sweep* phase.

A straightforward way to provide this optimization may be to substitute the handwritten functional implementation for suitable uses of *scan*, or expose it as a standard library function. However, this solutions have several limitations.

Firstly, the handwritten functional implementation makes use of two distinct and orthogonal parallelization strategies: the *scan-then-reduce* and *brent-kung*. It is reasonable to assume that in certain circumstances the compiler should be able to apply the two optimizations independently. For instance, for sufficiently small inputs just parallelizing the outer scan may have acceptable performance.

Moreover, in the course of implementing the functional version, a number of implementation details had to be decided, such as fixing the block size and the depth of the parallel scan tree, which materializes in the source code by influencing the amount of unrolled operations. These choices may not be optimal for many uses of the *scan* primitive. Indeed, in section 6 we will show how significant performance gains can be obtained by supporting a degree of variation in the algorithm used to generate optimized parallel scans.

The goal, therefore, is not just to express an optimization, but rather to *model an optimization space*. We achieve this by using a system of parametric rewrite rules, which the compiler can then use to generate the optimized code, be it on the basis of a set of heuristics, a user-defined specification, or an extensive process of automatic exploration.

scan-then-reduce(BS)

```
1 match scan(f,zero,input)
2 if input.size % BS == 0 ↦
3  let chunks = split(input, BS) |>
4   map(chunk => scan(f,zero,chunk))
5  let sums = chunks |> map(reduce(f, zero))
6  let scans = scan(f,zero, sums)
7  zip(chunks, scans) |> map(
8   (chunk, x) => chunk |> map(y => f(x,y)))
```

Listing 4: The *scan-then-reduce* rule parallelizes an abstract scan call.

4.1 Optimization via Rewrite Rules

Functional representations simplify the expression of program transformations using rewrite rules. A rewrite rule is a transformation that matches specified program patterns, rewriting them in accordance with a different pattern. This paper uses Elevate [5], a DSL for expressing rewrite rules in a compositional style. Primitive rules are specified via matching over program fragments, and then larger rules – known as *strategies* – are constructed by composing existing rules using a generic family of *combinator* functions.

In the syntax of the rewrite rules shown here, highlighted code refers to program fragments, while non-highlighted code is the rewrite rule logic – tasked with finding and replacing the such fragments. Rule logic can query type information, such as inspecting the known length of an array, be parameterized by numerical values, and perform simple numerical and logical computation.

Optimizations that are normally implemented ad-hoc within the compiler can therefore be expressed in this generic system. The rest of this section covers in detail the transformations generating *work-efficient parallel scan* implementation from an abstract high-level description.

4.2 Algorithmic Optimization

scan-then-reduce The first rule applied is *scan-then-reduce* (Listing 4). It matches a call to an abstract scan *scan(f,zero,input)*, replacing it with a parallelized version that uses the *scan-then-propagate* algorithm. The parameter BS expresses the block size for the parallel sub-scans. The rule requires the scan's input array to be divisible by BS. This is a necessary correctness check.

brent-kung As seen in section 3.2, the work-efficient block-level scan is an instance of *Brent-Kung* scan, which we also seek to express as a rewrite rule. As we have seen before, a *Brent-Kung* parallel scan recursively computes (*upsweep* phase) or consumes (*downsweep* phase) layers of a tree. In the functional formulation, this iteration is necessarily unrolled, as expressing it as a loop requires mutating the array that stores the tree.

brent-kung-step(BF)

```
1  match  scan(f,zero,input)
2  if input.size % BF == 0 ↦
3  // Upsweep
4    let partials = input |> split(BF) |>
5     map(chunk => reduce(f, zero, chunk))
6  // Recursion point
7    let rest = scan(f, zero, partials)
8  // Downsweep
9    zip(rest, partials |> split(BF)) |>
10    map((r, ps) => scan(f, r ps)) |> join
```

Listing 5: Each application of the rule adds one layer to the parallelization tree.

brent-kung(BF)

```
1  match  scan(f,zero,input)
2  if (log(BF, input.size) is whole) ↦
3   num_iterations := log(BF, input.size)
4   iterate(num_iterations)(
5    first(scan)(brent-kung-step(BF))
6   ) @  scan(f,zero,input)
```

Listing 6: The rule rewrites an abstract scan into a *brent-kung* parallel scan

This expansion is expressed with recursive rewrite rules. Listing 5 shows the rule for the recursive step. The rule is parameterized by the tree branch factor BF. The rule's body has three parts: first and last are the *upsweep* and *downsweep* phases. In between, the rule inserts an abstract scan call, acting as the recursion point. Based on *brent-kung-step* we can build the full *brent-kung* rule (listing 6), which expresses the iterative behavior. It inspects the size of the input array in the matched *scan* call to compute the depth of the parallel tree, determining the number of recursive applications of *brent-kung-step*.

The rule uses *combinators*: rules parametrized by other rules. The first combinator is `first(scan)(brent-kung-step(BF))`. It generates a rewrite rule that finds the first instance of `scan`, and rewrites it using `brent-kung-step(BF)`. The `iterate(num_iterations)` combinator then applies this repeatedly. This composition of combinators results in an iterative expansion of the *scan* supplied, each step adding one layer of *upsweep* before the scan and one layer of *downsweep* after the matched *scan* call, while simultaneously shrinking the leftover scan input by a factor of 2.

parallel-scan(BS, BF)

```
1 match scan(f,zero,input)
2  if BS % BF == 0
3  if input.size % BS == 0 ↦
4     (scan-then-reduce(BS);
5     first(scan)(brent-kung(BF))
6     ) @  scan(f,zero,input)
```

Listing 7: The *parallel-scan* rule combines parallelization strategies into the complete GPU scan.

parallel-scan We can now express the algorithm for the full GPU parallel scan by combining *scan-then-propagate* and *brent-kung* into a single rule, whose definition is shown in listing 7. The rule first applies the *scan-then-propagate*, followed by the *brent-kung* rule on the first instance of *scan* encountered.

5 Optimization Space Exploration

5.1 Expressing Scan Variants

As we have seen in the previous section, the *brent-kung* rewrite rule works by recursively expanding a call to scan into a parallel tree computation. Given block size BS and tree branching factor BF, the process is iterated $log_F(BS)$. As a variation, it is possible to terminate this expansion earlier, by parametrizing the *brent-kung* rule by TD, the maximum depth of tree expansion.

The overall optimization from *scan* to *parallel scan* now admits three possible parameters: the tree growth factor BF, the block size BS (obtained from the input parameter type), and the parallelization tree depth TD. These parameters delineate a space of possible optimizations. Given a value for the block size BS, the possible parallel tree depth ranges from $TD = 0$, which generates a fully sequential scan, to $D = log_{BF}(BS)$, yielding the canonical *Brent-Kung scan*. Intermediate values express hybrid versions, such as that shown in listing 8.

To see why such intermediate optimization points may be of interest, consider that on a GPU target each block maps to an OpenCL work-group. As the value of successive layers depends on that of preceding layers, these must be computed sequentially, introducing a synchronization point. Shrinking the parallel tree depth reduces the amount of synchronization necessary, but also trades away parallel computation for sequential work, growing exponentially as TD decreases by a factor of BF. For small reductions in TD this may be a positive trade-off.

5.2 Exploring Scan Variants

Finding the optimal (BF,BS,TD) triple is not a straightforward task: a sound choice of parameters requires knowledge of the target platform. For example,

```
1  void block_sum(float output[S], float input[S])
2  local float up1[8], up2[4], dn2[4], dn1[8];
3   parallel for block=0 to S/16
4    float* in = &input[16*block];
5    float* out = &output[16*block];
6    parallel for i=0 to 8
7     up1[k] = in[2*i]+in[2*i+1];
8    parallel for i=0 to 4
9     up2[k] = up1[2*i]+up1[2*i+1];
10   dn2[0] = 0;
11   for i=1 to 4
12    dn2[i] = dn2[i-1]+up2[i];
13   parallel for k=0 to 4
14    dn1[2*i]=dn2[i]; dn1[2*i+1]=dn2[i]+up1[2*i];
15   parallel for k=0 to 8
16    out[2*i]=dn1[i]; out[2*i+1]=dn1[i]+in[2*i];
```

Listing 8: Block sum with BF=2, BS=16, TD=2.

when targeting GPUs, the number of local thread used equals BF^{TD}. This value should be larger then the GPU's warp size to avoid needless stalls, but not too large, to minimize synchronization.

The compiler then generates and tests the variations, finding the best triple for the target architecture. In this paper, the search space is sufficiently small that it is practical to exhaustively explore it. Should the search space become large, one can alternatively use a more sophisticated search strategy, such as using an generic autotuner like OpenTuner.

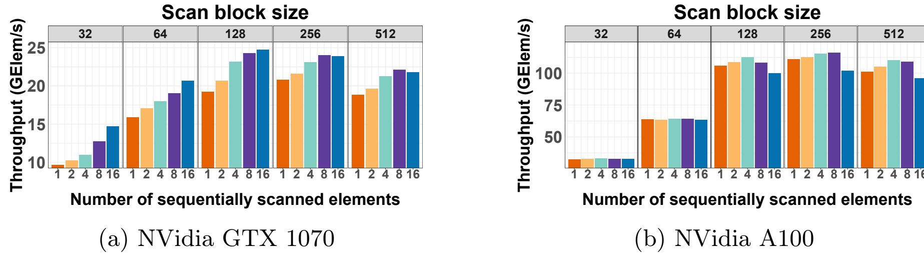
6 Evaluation

This section presents the paper's experimental results. All measurements are performed using the compiler's OpenCL CUDA back-end, targeting version 1.2 of the standard, driver version 10.2.185 and are run on NVidia GeForce GTX 1070 and NVidia A100 GPUs. All times refer to GPU computation time only. The *scans* compute the prefix sum of 32-bit floating-point values.

6.1 Performance of Scan Block Variants

Section 5.1 parametrized the *work-efficient parallel scan* generation by the triple of Branching Factor(BF), Block Size(BS) and Tree Depth(TD). This section presents the details of exploring this delimited space of possible variants.

We begin by fixing the value of $BF = 2$, which implies that the parallel iteration tree is a binary tree. For ease of presentation, we introduce the new parameter $SE \in [0, BS]$, indicating the number of Sequentially Scanned Elements.



(a) NVidia GTX 1070

(b) NVidia A100

Fig. 3: Throughput of automatically explored variations of *block_ scan*. Each version is parameterized by the scan block size (top) and by the number of elements that are sequentially scanned (bottom).

We wish $SE = 0$ to yield a fully parallel scan, while $SE = BS$ corresponds to an entirely sequential one. This desired behavior corresponds to constraining $TD = log_2(BS - SE)$. The range of sensible values for BS is shaped by the GPU's architecture, as it directly correlates to the amount of local memory required. Here, it ranges from 32 to 512, doubling each step. Fixing BS also allows determining the range of valid SE values: 1 to 16, also doubling each time.

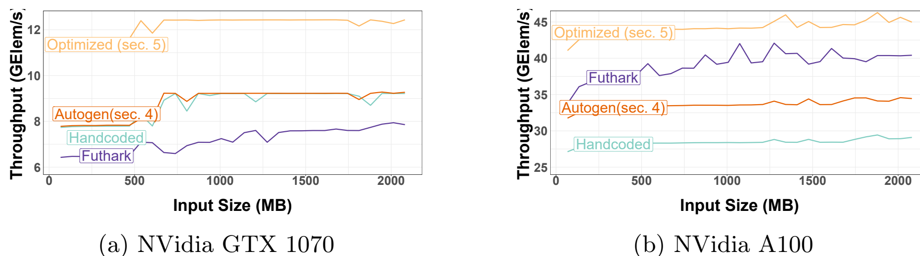
Figure 3 shows the results of exhaustively exploring the space delimited by these constraints. The source program is a prefix sum computation over an array of 25.6 million 32-bit floating-point elements. Executing the whole exploration takes approximately 40 minutes on the author's commodity hardware platform.

For both GPUs, the best versions have $SE > 4$. This is likely because higher values of SE imply a reduction in synchronization points, as these are required between parallel iterations. The trade-off is only beneficial with larger block sizes, as a small block with positive SE will lead to many of the warp's threads being idle. While the optimal values for SE and BS vary across GPU architectures, our approach can easily adapt to each platform's characteristics.

6.2 End-to-end comparison

The quality of our approach is evaluated by measuring the end-to-end performance of the generated code. This includes the block-scan as well as the subsequent propagation phases. The comparison covers both the baseline parallel scan shown in section 4 and the result of variant exploration in section 5 with two reference implementation. The first is a handwritten version provided by NVidia[7], and the second is the code produced by the Futhark compiler[8], a state-of-the-art data-parallel functional generator.

The results are shown in figure 4. Across both GPUs, the optimized version significantly outperforms both reference implementations. This is in contrast to Futhark, whose performance varies significantly across architectures. By expressing the optimization process via rewrite rules, our compiler can reliably generate high-performance code.



(a) NVidia GTX 1070 (b) NVidia A100

Fig. 4: End-to-end throughput of scan implementations. **Autogen** refers the code generation shown insection 4. **Optimized** refers to the best version obtained in section 5. **Handcoded** is the NVidia optimized version shown in [7], and **Futhark** is the version generated by the Futhark compiler[8].

7 Related Work

Data Parallel Functional Code Generators A recent trend in the design of high-performance code generators that use functional languages as inputs or internal representations. These include Lift [20, 21], Futhark [8], Single-assignment C [18] and Accelerate [3]. These compilers leverage the properties of a functional style to generate high-performance code for GPUs and other accelerators.

Rewrite Rules & Optimization Spaces The use of rewrite rules to express optimizations is well attested in the literature. We expressed our rewrite rules via the Elevate [5], which has also been similarly used for image processing applications [10]. The Spiral [16] compiler spearheaded using rewrite rules to optimize Digital Signal Processing applications in the SPL [22] language.

Petabricks [1] has been used to explore the design space of optimization for sorting algorithms. The use of an auxiliary language to model optimizations has similarities in Halide [17] schedules.

Parallel Scan There is a wide literature concerning the use of scan in parallel programs, starting from the seminal work of Blelloch [2]. Much work has gone into producing parallel implementation for the GPUs, from early CUDA implementations[7] to libraries such as CUDPP [19] and CUB [12]. Parallel scan is also a topic of relevance in the functional programming community. In particular [13, 4], which show algorithms to derive parallel scan implementations.

8 Conclusion

This paper presented a functional formulation of work-efficient parallel scan. We have decomposed it in a series of rewrite rules, modeling an optimization space. Exploring this space yields efficient implementations across GPUs from the same high-level source, consistently outperforming both an hand-written implementation and state of the art code generator, with up to 1.5x improvement.

Acknowledgements and Data Availability Statement

The datasets and code generated and evaluated in the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.19980176 [15].This work has been supported by the Engineering and Physical Sciences Research Council, grant number 1819353. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

References

1. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A language and compiler for algorithmic choice. ACM Sigplan Notices **44**(6), 38–49 (2009). https://doi.org/10.1145/1542476.1542481
2. Blelloch, G.E.: Scans as primitive parallel operations. IEEE Transactions on computers **38**(11), 1526–1538 (1989). https://doi.org/10.1109/12.42122
3. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: POPL-DAMP (2011). https://doi.org/10.1145/1926354.1926358
4. Elliott, C.: Generic functional parallel algorithms: Scan and fft. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–25 (2017). https://doi.org/10.1145/3110251
5. Hagedorn, B., Lenfers, J., Koehler, T., Gorlatch, S., Steuwer, M.: A language for describing optimization strategies. arXiv preprint arXiv:2002.02268 (2020). https://doi.org/10.48550/arXiv.2002.02268
6. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: CGO (2018). https://doi.org/10.48550/arXiv.2002.02268
7. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. GPU gems **3**(39), 851–876 (2007)
8. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In: PLDI (2017). https://doi.org/10.1145/3062341.3062354
9. Hinze, R.: An algebra of scans. In: International Conference on Mathematics of Program Construction. pp. 186–210 (2004). https://doi.org/10.1007/978-3-540-27764-4_11
10. Koehler, T., Steuwer, M.: Towards a domain-extensible compiler: Optimizing an image processing pipeline on mobile cpus. CGO (2021)
11. McCool, M., Reinders, J., Robison, A.: Structured parallel programming: patterns for efficient computation. Elsevier (2012). https://doi.org/10.1145/2382756.2382773
12. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. NVIDIA, Tech. Rep. NVR-2016-002 (2016)
13. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. PLDI (2007). https://doi.org/10.1145/1273442.1250752

14. Pizzuti, F., Steuwer, M., Dubach, C.: Generating fast sparse matrix vector multiplication from a high level generic functional IR. In: CC (2020). https://doi.org/10.1145/3377555.3377896
15. Pizzuti, F., Steuwer, M., Dubach, C.: Artifact and instruction for replication of experiments in the europar '22 paper titled: "generating work efficient scan implementations for gpus the functional way" (2022). https://doi.org/10.6084/m9.figshare.19980176
16. Puschel, M., Moura, J.M., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., et al.: Spiral: Code generation for dsp transforms. Proceedings of the IEEE (2005). https://doi.org/10.1109/JPROC.2004.840306
17. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.P.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: PLDI (2013). https://doi.org/10.1145/2499370.2462176
18. Scholz, S.: Single assignment C: efficient support for high-level array operations in a functional setting. J. Funct. Program. (2003). https://doi.org/10.1017/S0956796802004458
19. Sengupta, S., Harris, M., Garland, M., et al.: Efficient parallel scan algorithms for gpus. NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003 **1**(1), 1–17 (2008)
20. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. In: Fisher, K., Reppy, J.H. (eds.) ICFP (2015). https://doi.org/10.1145/2784731.2784754
21. Steuwer, M., Remmelg, T., Dubach, C.: Lift: a functional data-parallel IR for high-performance GPU code generation. In: CGO (2017)
22. Xiong, J., Johnson, J., Johnson, R., Padua, D.: Spl: A language and compiler for dsp algorithms. ACM SIGPLAN Notices **36**(5), 298–308 (2001). https://doi.org/10.1145/378795.378860