Utrecht University

Graduate School of Natural Sciences

Computing Science

# A Robust Multidimensional Parallel Scan Algorithm for Multi-Core CPUs

*Author:*
Rudolf de Kluijver

*First Supervisor:*
Prof. Dr. G.K. Keller
*Second Supervisor:*
Dr. M.I.L. Vákár
*Daily Supervisor:*
D.P. van Balen MSc

Master Thesis
July 2024

## Abstract

Scan operations are commonly applied in a wide variety of parallel applications and programming languages. Existing scan algorithms, for the CPU architecture, are either limited to one-dimensional input sequences or have some edge-case input shapes in which they lose their performance. The adaptive chained scan algorithm transformed the original chained scan with decoupled look-back to efficiently work on multi-core CPUs. This algorithm outperforms existing three-phase scan algorithms and offers flexibility in the number of working threads and has zero overhead compared to the sequential scan during single-threaded executions.

We present the assisting column-wise chained scan algorithm, which offers the same properties as the adaptive chained scan algorithm and performs efficiently and similarly regardless of the shape of the multidimensional input sequence. The algorithm is implemented and evaluated in Rust, by comparing it with the existing multidimensional scan implementation of Accelerate and the one-dimensional adaptive chained scan. Our results show that the assisting column-wise chained scan algorithm performs significantly better in edge-case scenarios at the cost of a slight performance loss in normal scenarios, which introduces a trade-off between robustness and maximal performance.

# Contents

# 1 Introduction

Scan algorithms, also known as prefix-sums, are commonly applied within a variety of applications, such as ordering algorithms or image processing. Taking a sequence of elements, an initial value and a binary operator, the scan algorithm generates an output sequence, in which the operator consecutively combines the elements from the original sequence. For example, a sequential scan can be implemented as follows:

```
function SEQ_SCAN(input, output, size, initial)
    acc ← initial
    for i ← 0 .. size do
        acc ← acc ⊕ input[i]
        output[i] ← acc
```

Although this sequential version works fine in many applications, its performance is limited when it comes to processing large input sequences in a short amount of time. If the binary operator is associative, however, the scan operation can also be executed in parallel over the input sequence. As multi-core processing on *Central Processing Units* (CPU) gradually became more powerful with each new generation and the *Graphical Processing Units* (GPU) got introduced, the parallelization of these types of algorithms became a popular topic in this field of research. The scan algorithm became a primitive building block for many parallel applications [2, 1], and several parallel versions of the algorithm were introduced [7], which improved the overall performance and efficiency when processing a larger number of data elements.

An example of the application of the scan operation in other parallel algorithms is array compaction/filtering. The goal of such a filter operation is to determine for each element in an input array, whether it satisfies some given property (provided by the programmer), to create a single output array consisting of the correct elements that satisfied the given property. Imagine providing multiple properties to the filter operation at once, this could quickly reduce the total size of the input array to a much smaller array containing just the relevant elements. In that case, much fewer elements have to be considered by the remainder of the algorithm, which potentially increases the overall performance of the program, at the cost of a single scan over the input elements.

While the parallelization of the scan algorithm over one-dimensional sequences has been studied for years, there is almost no research published that focuses the scan operation over multidimensional data structures (e.g. matrices). Existing implementations of these type of scans, for example in Accelerate [5], are repeatedly applying a sequential scan to each row in the inner-dimension of the data set. This approach, however, can be inefficient when the input data set has a skewed shape. In the example of a matrix, this indicates that there is either a small number of rows combined with a large number of elements on each row, or the other way around. In these types of scenarios, the

performance of the described multidimensional scan is often limited by two main factors. One of these factors is an incorrect distribution of work over the available threads and the other factor is the limited memory bandwidth. These limitations especially affect CPUs, as these devices have a small number of threads and significantly less memory bandwidth compared to the GPU. Furthermore, the CPU potentially runs other background tasks during the execution of the scan operation, while the GPU is purely dedicated to these type of computations.

A recently published paper, called 'zero-overhead parallel scans for multi-core CPUs' [6], introduces improved versions of existing parallel scan algorithms. Up until then, most parallel scan algorithms for the CPU had some overhead compared to the sequential scan performance, meaning that these algorithms were less performant on single-threaded executions. Therefore, the paper introduces new versions of these algorithms with equal performance to the original, while also offering the property of zero overhead. The main contribution is the introduction of the adaptive chained scan algorithm, for which the decoupled look-back algorithm [15], originally designed for the GPU, is translated into a suitable version for the CPU architecture. Benchmarks over one-dimensional sequences show promising results, as the adaptive chained scan algorithm outperforms the existing scan algorithms in most scenarios. However, this algorithm is currently unable to handle multidimensional input data.

Therefore, the goal of this thesis project is to develop a parallel scan algorithm for the CPU, that is capable of efficiently handling multidimensional data structures, regardless of their shape. This multidimensional algorithm uses concepts shown in the adaptive chained scan algorithm and should ideally be able to run with zero overhead during single-threaded executions. Furthermore, this algorithm is expected to be robust against skewed input shapes and should perform similar to the existing multidimensional scan of the Accelerate framework [5] in the other input scenarios. More specifically, the following research questions will be studied:

- How can the concepts of the zero-overhead adaptive chained scan algorithm be used to create a parallel scan algorithm for multidimensional data structures?
- How can this multidimensional scan algorithm be designed to perform the execution efficiently over the available threads, regardless of the shape of the data structure?
- What is the expected impact on array-based languages, like Accelerate, when implementing this multidimensional scan algorithm into their framework?

# 2 Background

This section contains the background information used to get an understanding of multidimensional parallel scans and the possible approaches of solving this problem. Section 2.1 goes into more detail about the concept of the scan operation, describes what factors are important towards an efficient scan algorithm and discusses the concepts of multidimensional scan operations. The zero-overhead parallel scan paper, introducing efficient and flexible parallel scan algorithms over one-dimensional data structures, is explained in more detail in Section 2.2. Finally, Section 2.3 discusses some basics of the Accelerate language, as well as an overview of the existing scan algorithms in this framework.

## 2.1 Scan operation

The scan operation is a fundamental primitive for many parallel algorithms, which makes it one of the most studied patterns in parallel applications. Back in 1989, Blelloch [1] introduced the scan as a building block for a variety of applications. The year after, he published a paper [2] showing the usage of the scan in several practical algorithms such as Radix-Sort and Quicksort, which further demonstrates the efficiency of applying the prefix-sum algorithm in parallel applications.

Given a sequence of elements, a binary associative operator $\oplus$ and a starting value $n$, a scan operation produces an output sequence of equal size to the input, in which each element represents the reduction of all prior elements in the original input sequence. The result of a scan operation can be either *inclusive* or *exclusive*. An inclusive scan means that the $i^{th}$ input element is incorporated into the $i^{th}$ output reduction, while an exclusive scan means that the $i^{th}$ input element is not incorporated into the $i^{th}$ output reduction. For example, the exclusive result of the scan operation described, using the input array $x = [x_0, x_1, ..., x_k]$, would be:

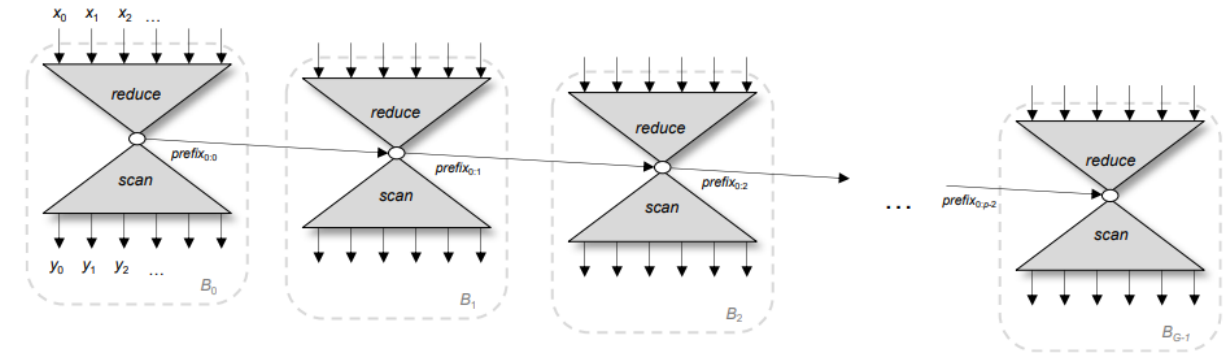
$$[n, n \oplus x_0, n \oplus x_0 \oplus x_1, ..., n \oplus x_0 \oplus ... \oplus x_k]$$

Common implementations of parallel scan operations are often making use of the classic three-phase scan strategies or a modified version based upon these three-phase scans, including *scan-then-propagate* [16] and *reduce-then-scan* [7]. In this context, *three-phase* indicates that the scan algorithm operates in three individual phases. Each thread is required to finish its work within the current phase, before the algorithm proceeds to the next phase. Other papers specifically focus on a linear scan approach [18, 15] and propose a parallel scan algorithm for the GPU called the *chained-scan*. The chained-scan algorithm applies a *single-pass approach*, which indicates that the scan operates in a single phase. In comparison to the three-phase scans, the chained scan has no restriction that requires all threads to be finished, before proceeding with the execution.

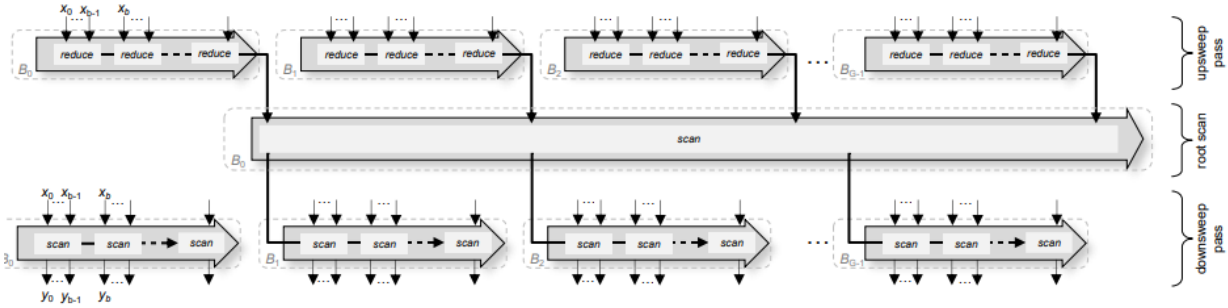
Most of the existing research on scan algorithms is specifically targeting the GPU hardware. Compared with the CPU, GPUs offer a much higher computation density and off-chip memory bandwidth, because of their organisation of the processing cores. Therefore, most research contains parallel applications that are suitable for the GPU architecture, but many of these can also be modified to work on multi-core CPUs. This is shown in the zero-overhead parallel scan paper [6], which we'll briefly cover in Section 2.2 and is used as a starting point for this research project.

### 2.1.1 Efficiency

An important property whilst working on a scan algorithm is to keep the number of *passes* as low as possible. In this context, a single pass is a series of operations that should be completed for all elements in the input sequence, before the algorithm can proceed with the next set of operations. Figure 1 shows an *execution diagram* of two parallel scan algorithms mentioned before, namely the single-pass *chained scan* [18] and the *reduce-then-scan* [7, 9] algorithm. Considering the diagram of the reduce-then-scan algorithm (Figure 1b), we can determine three separate passes. The first pass is the reduction of all elements in a block to a single value. After all of these reductions are completed, the next pass is executed in which a single thread scans over the reduced values from the previous pass. When this scan is completed, the third pass incorporates the resulting values from the second pass in the final scan result. In comparison, the chained scan algorithm (Figure 1a) combines the reduce and scan operations into a single pass, and transfers the required reduction value directly between the working threads ($B_0$ until $B_{G-1}$). In general, reducing the number of passes in an algorithm increases its performance, as there is less total waiting time between the consecutive passes.




**(a)** Single-pass *chained scan* ($\sim 2n$ global data movement)




**(b)** Three-kernel *reduce-then-scan* parallelization ($\sim 3n$ global data movement)

***Source:*** Original from Merrill and Garland [15]

**Figure 1:** Example execution diagram of parallel scans among G thread blocks

Reducing the total number of *memory accesses* is another aspect for a well-performing scan algorithm. Although most of the modern processing units are highly optimized to quickly perform binary operations, they are generally bound by the high cost of memory accesses, such as reading input data and writing results back to global memory. Therefore, minimizing the number of global *data movements*, necessary for the execution of the scan, is a good objective for increasing the total efficiency and performance of the algorithm. In the case of scan operations, the optimal solution requires only a single read of the input values from memory and a single write of the results back to memory, resulting in an optimal number of $2n$ global data movements. This property is achieved by the sequential version of the prefix scan algorithm, which is used as the baseline measurement during our benchmarks.

Parallel versions of the prefix scan are optimized to get as close as possible to this optimal solution. The parallel single-pass chained scan algorithm (Figure 1a) makes use of $G$ thread blocks, which are each assigned a tile of the input data, labelled $X_i$. Furthermore, there is a serial dependence chain between the thread blocks, indicated by the arrows labelled $prefix_{0:i}$. This dependency arises because the reduction results of one thread block needs to be transferred to the next thread block in line to complete the scan operation. At the end of each thread block, the results are written back to memory, labelled with $Y_i$. This results in a global data movement of $\sim 2n$ ($n$ reads, $n$ writes, $(G-1)$ inter-thread communications), which is close to the optimal solution. In comparison, the parallel *reduce-then-scan* algorithm (Figure 1b) is less efficient in regard to the number of global data movements. In the execution diagram we can see that the values are read twice from memory, namely in the reduce phase as well as the scan phase. Furthermore, the results are written back to memory on completion of the algorithm and some additional data movements are necessary to send the results between the different phases, using a small array of size $G$ (= number of threads). This adds up to a total number of $\sim 3n$ global data movements, which is less efficient than the number of data movements of the *chained scan* approach.

### 2.1.2 Multidimensional scans

Nowadays, many research papers contain optimised parallel scan algorithms for one-dimensional input sequences. However, as far as I'm aware, none of these papers really focus on the multidimensional arrays of data (e.g. matrices) and the application of a prefix scan algorithm in these scenarios. Šinkarovs and Scholz [17] propose a scan operation, which reshapes incoming (one-dimensional) data into multidimensional arrays. The shape of this array is in turn used to guide the process of the scan operation, such that the compiler is able to create efficient parallel code by applying multiple sequential scans. This algorithm still handles just one-dimensional data sequences, but the guidance of the traversal based on the array shape is a basis for further research.

A common approach for computing the prefix sum over matrices of data, is to store the elements in a *flat array* and use the respective shape to assign a sequential scan upon each row in the innermost dimension. The shape is used to determine the number of elements on each row and the total number of rows with their respective starting positions. This guidance based on the array shape is in some sense similar to the idea of Šinkarovs and Scholz, apart from the fact that they

transfer one-dimensional data into a multidimensional array. An example execution of the prefix sum operation over a given matrix is shown in Figure 2, in which we consecutively scan all rows in the innermost dimension.

$$\underset{\textit{(input)}}{\begin{bmatrix} x_0 & x_1 & x_2 \\ x_3 & x_4 & x_5 \\ x_6 & x_7 & x_8 \end{bmatrix}} \implies \underset{\textit{(output)}}{\begin{bmatrix} n \oplus x_0 & n \oplus x_0 \oplus x_1 & n \oplus x_0 \oplus x_1 \oplus x_2 \\ n \oplus x_3 & n \oplus x_3 \oplus x_4 & n \oplus x_3 \oplus x_4 \oplus x_5 \\ n \oplus x_6 & n \oplus x_6 \oplus x_7 & n \oplus x_6 \oplus x_7 \oplus x_8 \end{bmatrix}}$$

**Figure 2:** Inclusive prefix sum over a matrix

Although this solution works fine on this small matrix example, there are some downsides to consider, especially when the input data set grows. First, the shape of the input sequence can affect the performance of the algorithm, such that the processor is not used to its full potential. Consider a multidimensional input array in which there is a small inner dimension. In this case, the processor scans the rows on the innermost dimension, but is unable to fully utilize the available processing power, as there is only a small number of elements available on each row. If the input array contains a large number of rows, this can gradually build up and reduce the total throughput of the algorithm. Secondly, some scan algorithms require an intermediate array to store their values. On top of that, the results of each row in the inner dimension should be stored separately as well, which means that there is a risk of having a large number of memory accesses during the execution of the algorithm. As we have seen that memory accesses are expensive in terms of performance, this can significantly reduce the final performance of the algorithm.

These are some of the challenges that need to be considered, when taking the prefix scan into the multidimensional context. The aim of this thesis project is to find a suitable algorithm for multidimensional prefix scans on (multi-core) CPUs. The algorithm should be applicable to most languages and frameworks that support multidimensional data structures. For the evaluation of this thesis project, the algorithm will be implemented using the Rust programming language [12] and tested against the existing scan operation of Accelerate [5].

## 2.2 Zero-Overhead algorithm

The single-pass parallel prefix scan algorithm with decoupled look-back, by Merrill and Garland [15], improved the existing research on parallel GPU scan algorithms, as it alleviates the problem of serial dependencies, increases the memory bandwidth usage and reduces the number of global data movements to $\sim 2n$. Taking this concept of the GPU algorithm and transforming it to efficiently work on multi-core CPUs is the main contribution of the zero-overhead paper [6]. While many of the existing parallel algorithms suited for the CPU architecture already have a low number of passes and global data movements, they still contain a constant *factor of overhead* when comparing it to the sequential scan on single-threaded executions. This overhead exists, as these parallel algorithms traverse the input data multiple times, while the sequential version traverses the data only once.

The improved parallel scan algorithms discussed in the paper, not only resolve the constant factor of overhead, but also provide a more flexible solution for parallel execution of the algorithm. As many existing algorithms require knowledge about the available number of threads beforehand, the zero-overhead algorithms are able to start in a sequential mode and adapt whenever more threads become available throughout the execution. Especially in the case of the CPU, this can be highly beneficial. Threads are sparse in these type of devices and this option to adapt makes it possible for a thread to assist, whenever it completes its current task.

The paper also introduces adapted versions of the three-phase scans, called *assisted reduce-then-scan* and *assisted scan-then-propagate*, which are outperforming the original versions of these algorithms. During the experiments, however, they find that the adapted version of the chained scan [15] algorithm performs even better in most scenarios. Hence, this thesis will be focusing on applying the concept of this *adaptive chained scan* algorithm for the multidimensional prefix scan, as it provides the most promising results for further research.

### 2.2.1 Parallel chained scans on CPUs

The chained scan approaches, described by Yan et al. [18] and Merrill and Garland [15], are originally designed for massively parallel GPUs. In contrast to CPUs, GPUs make use of thread blocks which can easily communicate with each other at low overhead, and contain significantly more register memory to store all elements within a block. However, the number of registers on the CPU is quite limited, and to be able to keep a large enough block size to reduce synchronization overhead, the elements are stored in L1 cache instead. The memory access cost of L1 cache is lower than the access cost of global memory, but the available storage is relatively small and differs between different processing units. This means that the maximal block size, used during the execution of the algorithm, is now depending on the specific processor and the size of each element in the array.

The look-back property of the chained scan means that threads are able to determine the required prefix value themselves, instead of idling until the result of the predecessor comes in. The incoming data is split into blocks of a fixed size, after which the parallel threads claim the blocks from left to right. A thread reduces its block to a single value, called the *aggregate*, which is directly shared. Then it requires the *prefix* value of the previous block, which is the last value in the block after scanning, to determine its starting value and perform the final scan. Whenever the prefix value of the predecessor is not yet available, the thread block can use the aggregate value and determine the prefix value themselves by continuing the look-back to the next predecessor in line. This look-back continues until a block with a prefix value is found, and the final scan can be performed using the calculated value.

Therefore, a descriptor object is stored per block, to keep each of the threads synchronized during execution. The descriptor object contains the following information:

- The status flag, which can be either:

  **X** - The field is initialised, but there is no progress made
  **A** - The aggregate value is available

**P** - The prefix value is available

- The aggregate value
- The prefix value

All of the descriptor objects are initialised with status flag **X**. Regarding the look-back iterations, it means that we take the aggregate of the predecessor when the status flag is **A** and the prefix value if the status flag is **P**, after which the look-back finishes.

### 2.2.2 Adaptive chained scans

To achieve zero-overhead on single-threaded execution, the operation of the chained scan algorithm (Section 2.2.1) is slightly changed, resulting in the *adaptive chained scan* algorithm. However, there is one restriction on chained scans to consider, namely the order of block processing. As each block depends on the aggregate values of its predecessors, they need to be processed from the left to the right. Therefore, the adaptive chained scan algorithm starts in a sequential mode on the leftmost block and switches to parallel mode when another thread joins the execution. When a thread claims the next block in line, it directly checks whether the prefix value of the previous block is already available. In that case, the thread directly performs the scan operation over its block and skips the reduce step and the aggregate calculation. However, when the prefix value is not yet available, the thread performs the usual operation of the chained scan as we've seen before. In single-threaded scenarios, the prefix value of the predecessor will always be available and the current block can directly be scanned using that value, reducing the total number of operations. Hence, when executed on a single core, this adaptive chained scan will have no overhead compared to the sequential counterpart.

## 2.3 Accelerate

As discussed earlier, parallel scans are applied in many array-based languages, one of them being the *Accelerate* [5] language. We introduce this language in more detail and discuss their current method for one-dimensional and multidimensional scan operations.

Accelerate is a functional embedded language in Haskell, which is used for high performance array computations through parallelism. The language is compiled to efficient (parallel) code for multi-core CPUs or optimized CUDA code for GPUs [5]. It uses a runtime compiler, which means that compilation happens on-the-fly during execution.

The Accelerate library contains many combinators, which can be combined to create a variety of parallel operations. These operations are further optimized by the compiler [13] as it combines primitives to reduce the total amount of computations and removes intermediate data-structures, which can significantly increase the performance of programs. Many of the functions available in Accelerate will look familiar to Haskell programmers. For example, the function that calculates the dot product of two given vectors can be defined as follows:

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 $ zipWith (*) xs ys
```

The `Acc` in the type definition is used as an indication that the calculation can later be executed on the selected device. The `Vector` type indicates an one-dimensional array and the `Scalar` type indicates a single value. The functions `fold` and `zipWith` are in Accelerate applied to arrays of data. Although they look similar to the corresponding Haskell functions on lists, there is a main difference, as these functions in Accelerate can be executed in parallel.

To compile this code to either the CPU or the GPU, Accelerate makes use of an *internal representation* (LLVM IR). The LLVM backend of Accelerate [14] contains a seperate LLVM compiler for native CPU code and GPU code. These compilers support many different algorithms including the scan algorithm, which is further explained in Section 2.3.1

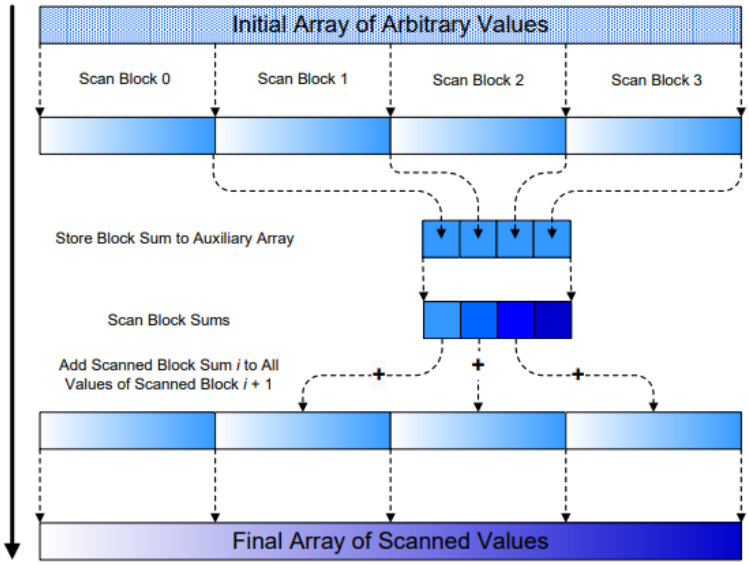
### 2.3.1 Scan operation

For the evaluation of this research project, the proposed multidimensional scan operation is compared against the current multidimensional scan in Accelerate. Therefore, it is useful to have a overview of the existing scan functionality in this language. The Accelerate library currently offers support for both one-dimensional array operations, as well as multidimensional arrays. These arrays are stored in memory as *flat arrays*, with separately stored *shape* information. This is represented using the `ArrayR` data type, which contains information about the dimensions and the data layout/type of the corresponding array:

```
data ArrayR a where
  ArrayR :: { arrayRshape :: ShapeR sh
            , arrayRtype  :: TypeR e
            }
         -> ArrayR (Array sh e)
```

In the current implementation of scan is a clear distinction between one-dimensional and multidimensional input arrays. One-dimensional arrays of a small size are scanned using a sequential operation, while arrays of a larger size, are scanned using the *scan-then-propagate* algorithm. This algorithm, as visible in Figure 3, first splits the input data into fixed size blocks. Then it reduces all blocks to a single value, using a parallel execution, and stores the resulting values in a temporary array. A single thread scans over this array to compute all of the aggregate values. Finally, it combines the aggregates with the respective blocks in parallel, resulting in the final scanned array. In total, this process takes $\sim 4n$ global data movements, which is far from the optimal solution.

The scan operation over multidimensional array structures is currently implemented as a sequential scan over a row, which executes in parallel over the outer-dimension. This means that multiple rows are scanned in parallel, but each of these rows is handled by a single thread. Therefore, it scans each row sequentially, instead of multiple threads working together on a single row as seen with, for example, the scan-then-propagate algorithm. The downside of this approach is, that the




***Source:*** Original from Harris et al. [10]
**Figure 3:** The parallel scan-then-propagate algorithm ($\sim 4n$ global data movement)

shape of the input array can heavily affect the performance of the scan algorithm. We consider two *edge case scenarios* that can be inefficient for the performance of this algorithm:

- A small matrix consisting of many rows and few elements per row
- A wide matrix consisting of few rows and many elements per row

For the first edge case, consider an input array with dimension $3 \times 10.000$ elements. The outer dimension of this array is 3, which means that the algorithm starts at most 3 threads in parallel, even in situations where more threads are available for the computation. A single thread, therefore, performs the scan operation sequentially over a row of 10.000 elements. As most modern CPUs contain at least 8 threads, it would be more efficient to have multiple threads working in parallel over the elements of a row, significantly improving the performance of this scan operation in these scenarios. The second edge case can be inefficient, as each thread is being assigned to a single row consisting of a small number of elements. Therefore, each thread will only request a small chunk of elements from memory for its respective row, instead of larger chunks containing the number of elements of the fixed block size. Therefore, the algorithm performs more (small) memory fetches throughout the entire scan operation than strictly necessary, which is expected to result in some performance loss.

Hence, the goal of this thesis project is to find a better solution to this existing scan operation over multidimensional array structures and compare their performances.

# 3 Multidimensional Parallel Scan

As we have seen the concepts and results of the adaptive chained scan algorithm (Section 2.2.2) on one-dimensional arrays, the question about how to apply these concepts in the multidimensional context and create an efficient multidimensional parallel scan algorithm still remains. A key property of the single-pass chained scan approach, in which input sequences are divided into blocks of a fixed size, is the linear dependency that exists between the consecutive blocks. The aggregate and prefix values of the predecessors are a necessity to complete the computation over the elements in the current block. Therefore, the order in which blocks of data are assigned to the working threads should ensure that all blocks of a row are assigned in their original order.

The goal of the multidimensional scan algorithm is to scan the input sequences over the innermost dimension, as shown in Section 2.1.2. Considering a flat array representation in memory, we can translate multidimensional input sequences in terms of a matrix, in which the innermost dimension represents a single row and the product of the other dimensions represents the total number of rows. The columns in this matrix are determined by the number of blocks necessary to fit a single row of the innermost dimension, which depends on the size of the innermost dimension and the fixed size of each block.

Based on these aspects, we consider two main patterns in which the blocks can be assigned, namely a row-wise and a column-wise block assignment pattern, which are discussed in Section 3.1 and Section 3.2 respectively. Then, we explain the design of our multidimensional scan algorithm in Section 3.3. The implementation of the algorithm in Rust [12] will be explained in Section 3.4.

## 3.1 Row-wise block pattern

Assigning the blocks in a row-wise pattern to the working threads is a reasonable approach for a multidimensional scan algorithm, as this pattern ensures that all blocks of an inner-dimensional row are assigned in their original order. This pattern is shown in Figure 4, which shows a multidimensional input sequence that consists of three rows on the innermost dimension, each of which is divided into three fixed size blocks. The consecutive rows are stored serially in memory because of the flat array representation discussed earlier. Therefore, considering the last elements of a row to be contained in block $n$, the first elements of the next row are contained in block $n + 1$. The same holds for the order of the blocks on a single row, which makes the row-wise assignment pattern a constant increment of the block index by 1, until all input data is processed.

The concepts shown in the zero-overhead paper [6] can be applied within this row-wise multidimensional scan algorithm, as it essentially is a collection of one-dimensional sequences, as long as we keep track of the start and end index of each row. Application of the adaptive chained scan (Section 2.2.2) principle, results in no overhead compared to the sequential scan performance on single-threaded executions. As the input data is multidimensional, we need to consider the situation in which the first block of a (new) row is being processed, as this effects whether the aggregate values of the predecessors can be ignored. Therefore, knowledge is required about the number of blocks necessary to contain all elements of a row and the total length of a single row. The length is

Blocks

| Rows | | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| | 4 | 5 | 6 |
| | 7 | 8 | 9 |

**Figure 4:** Row-wise block assignment pattern

used to determine the size of the last block on each row, as the number of remaining elements can be smaller than the maximal allowed number of elements in a single block.

Increasing the number of working threads, however, increases the chance of threads performing multiple iterations within the look-back phase to determine the required prefix value. All of the threads are working on consecutive blocks and are therefore likely to be working on the same row. However, if the input sequence contains many rows, the threads can also perform their scan operation on separate rows instead, without interference of other threads on predecessor blocks. Therefore, we consider an alternative block assignment pattern, which is further explained in Section 3.2.

### 3.2 Column-wise block pattern

Assigning the blocks in such a way that the interference between different threads working on consecutive blocks is minimized, is the goal of the column-wise block assignment pattern. The assignment order is visible in Figure 5, which again represents a matrix consisting of three rows on the innermost dimension each of which is divided into three fixed size blocks. In contrast to the row-wise pattern, we first assign the first block of each row, followed by the second block of each row, continuing until all input data has been assigned and processed. This way all blocks of a row are still processed in their original order, while the working threads are spread over the available rows, reducing the interference of threads on consecutive blocks.

Blocks

| Rows | | | |
|---|---|---|---|
| | 1 | 4 | 7 |
| | 2 | 5 | 8 |
| | 3 | 6 | 9 |

**Figure 5:** Column-wise block assignment pattern

Consider a situation where the number of rows is significantly larger than the number of working threads. Furthermore, assume that the $n^{th}$ block of all rows has been assigned to the working

threads, meaning that the $(n+1)^{th}$ block of the first row is the next block to be assigned by the column-wise pattern. Whenever this next block is being processed, the work on its predecessor is almost certainly finished, as the number of rows is significantly larger than the number of threads. In that case, the required prefix value can directly be read from the block descriptor array by applying the adaptive chained scan principle, as seen in Section 2.2.2.

Theoretically, all threads spend less time calculating the prefix value themselves using look-back iterations, as the prefix value of the predecessor is mostly directly available. Therefore, we expect a slight performance improvement compared to the row-wise block assignment pattern.

## 3.3 Multidimensional algorithm

Now that both the row-wise block pattern (Section 3.1) and the column-wise block pattern (Section 3.2) are discussed, we can try to combine the strengths of these two patterns within a single multidimensional parallel scan algorithm. Ideally, the algorithm performs the scan operation efficiently, regardless of the number of working threads and the size and shape of the input data, including the edge case input sequences of either extremely small or wide matrices as described in Section 2.3.1. First, the design and the functionality of this algorithm is explained in more detail. Followed by an example state of the algorithm after execution of the scan operation, to explain the functioning of the algorithm in practice.

### 3.3.1 Assisting column-wise chained scan

Preliminary benchmarks between the row-wise chained scan and the column-wise chained scan, showed some advantages and disadvantages of both of these patterns. In fact, input scenarios with an increasing number of working threads are most of the times performing equally or better using the column-wise pattern instead of the row-wise pattern. Single-threaded executions, on the other hand, show some slight overhead when using the column-wise pattern, while the row-wise pattern is able to perform with zero overhead. The comparison between these two patterns is further discussed in Section 4.2.

The assisting column-wise chained scan algorithm introduces a combination between the performance of the column-wise pattern in multi-threaded scenarios and the zero-overhead property of the row-wise pattern. The algorithm is visualized using a flowchart in Figure 6. A separate conditional is included to determine whether the size of the innermost dimension is smaller or equal to the fixed block size. In that case, depending on the size of the innermost dimension, at least one or optionally multiple rows can fit within a single fixed size block. This potentially reduces the total number of blocks and block assignments to the working threads, resulting in less atomic increment instructions of the current block index.

In general, many parts of the algorithm are similar to the chained scan and adaptive chained scan algorithm, discussed in Section 2.2. The working threads still need to be synchronised using an array of descriptor objects, containing the status flag, the aggregate value and the prefix value for each block. The main difference, however, is the way in which the individual data blocks are

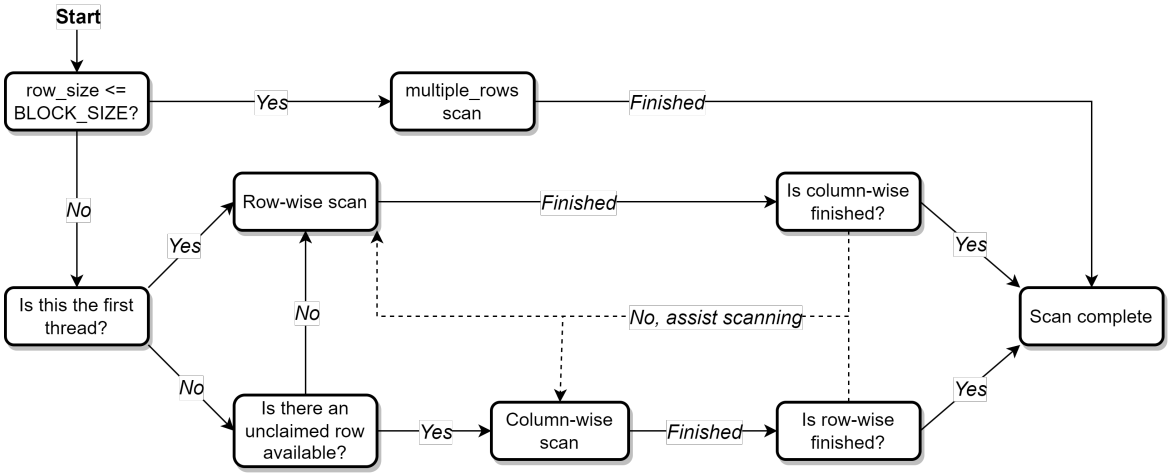



**Figure 6:** Flowchart assisting column-wise chained scan algorithm

assigned to the working threads. The first thread, which initially claimed the scan task, starts processing the blocks via the row-wise pattern, which ensures the zero-overhead property on single-threaded executions. The assisting/parallel threads will instead process the blocks via the column-wise pattern, as we've seen that this pattern is mostly beneficial during multi-threaded executions. The column-wise scan can only be applied if there's an unclaimed row available, as otherwise keeping track of assigning and processing all blocks of input data would be difficult. When multiple threads start working on the same task, the first thread detects the parallelism and finishes its current row in row-wise order, after which it assists with the column-wise scan. The same holds for the parallel threads, whenever the column-wise scan is finished before the first thread could finish its current row, they start assisting with the row-wise scan. We present the pseudocode for the assisting column-wise chained scan algorithm in Algorithm 3, together with the block assignment loop in Algorithm 2. These implementations will be further discussed in Section 3.4.

This design of the algorithm gives us a few properties. First of all, threads are prevented from idling for longer periods of time, as they're able to calculate the required prefix value themselves during the look-back phase and assist either the row-wise or column-wise scan upon completing their own task. Furthermore, the algorithm is able to switch to a parallel execution without any additional synchronisation overhead, which makes the algorithm able to work with a variable number of working threads throughout the execution. This is especially useful on a CPU, as available threads are usually sparse on these type of devices and other background tasks may finish during the process of the scan algorithm, after which the finished thread can assist the scan operation. Lastly, the multidimensional scan algorithm is also able to handle one-dimensional input sequences. The row-wise chained scan algorithm (Section 3.1), in which the algorithm starts , is practically equal to the adaptive chained scan algorithm (Section 2.2.2). Therefore, our performance on one-dimensional sequences is expected to be close to the performance shown in the zero-overhead paper [6].

### 3.3.2 Execution example

Suppose we have a small multidimensional input matrix, consisting of three inner dimensional rows that each can be divided into three fixed size data blocks. The assisting column-wise chained scan algorithm is applied on this input sequence to calculate the prefix sum using three parallel threads. Figure 7 illustrates a possible execution state after the algorithm is finished. The numbers represent the order in which each of the blocks is assigned to the workers and the colours represent the different threads that processed these blocks of data. We can identify the following stages:

- Thread 1 (yellow) starts the computation in row-wise order, as only a single thread is currently working on the task.
- Thread 2 (green) joins the computation, while thread 1 is working on the second block. Recall that assisting threads will only start processing blocks in column-wise order, whenever an unclaimed row is available. The second row is currently unclaimed, as thread 1 is still active on the first row. Therefore, the first block of the second row is assigned to the assisting thread, after which it continues processing the other blocks in column-wise order.
- Thread 1 detects the parallelism of the assisting thread upon completion of the second block, but still needs to finish the first row in row-wise order. Therefore, the third block of the first row is now assigned to thread 1, after which it assists the parallel threads with the column-wise order scan.
- Thread 3 (blue) joins the computation when thread 2 is already processing the data in column-wise order. Therefore, it directly starts assisting the column-wise process.

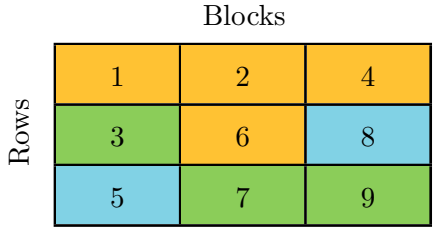



**Figure 7:** Example order of block assignment during a scan operation. The colours represent the different threads processing the data blocks.

### 3.4 Rust implementation

The implementation of the algorithms in Rust [12] can be subdivided into three main components: the data structure, the block assignment loop and the scan algorithms. Each of these will be discussed in more detail in the following subsections. The main focus will lie on the assisting column-wise chained scan algorithm, as both the row-wise chained scan and the column-wise chained scan are also embedded within the design of this algorithm. The full implementation of the algorithms including the benchmarks can be found on Github[1].

#### 3.4.1 Multidimensional data container

To correctly execute the parallel scan algorithms, two things are required besides the data itself, namely knowledge about the shape of the multidimensional data and the fact that the data itself needs to be linearly stored in memory, also known as the *flat array* representation. We implemented a minimal structure definition called `MultArray`, as shown in Listing 1, containing a reference to the data and the corresponding (multidimensional) shape of the data. The `MultArray` takes the shape information and the element type $T$ on initialisation, and uses this to allocate a fixed sized array for elements of type $T$ on the heap. The `Box` type is then used to create a reference to this location on the heap, which can easily be shared between the different functions.

```
pub struct MultArray<T, const N: usize> {
  data: Box<[T]>,
  shape: [usize; N],
}
```

**Listing 1:** Multidimensional data structure in Rust

#### 3.4.2 Parallel block assignment

The different data blocks are assigned to the working threads using a parallel loop. Recall that an important property of the chained scan algorithm is that all blocks need to be processed exactly once and in their original order. Therefore, a global variable *work_index* is used, which can be updated by the threads to determine the current index of the block to be processed. As multiple threads may be simultaneously working on the same task, we use atomic instruction to safely update the *work_index* variable.

Both the row-wise chained scan and the column-wise chained scan determine their next block index by atomically incrementing the *work_index*. The pseudocode for this loop is shown in Algorithm 1. The row-wise scan uses the number of blocks on a single row, to determine the current partition of elements to process. Whereas, the column-wise scan does an equal calculation, but uses the total number of rows instead. This difference is also visible in Algorithm 3, in which the

[1] https://github.com/Rudolf-UU/multi-dimensional-parallel-scan

*row_idx* and *column_idx* variables are determined differently, while the remainder of the algorithms is similar.

**Algorithm 1** Parallel block assignment

```
function BLOCK_ASSIGNMENT_LOOP(work_size, uint32* work_index, scan_function)
    loop
        block_idx ← ATOMIC_FETCH_ADD(work_index, 1, ordering::relaxed)
        if block_idx >= work_size then break
        scan_function(block_idx)
```

The assisting column-wise chained scan, however, needs to keep track of the current index in both the row-wise order and the column-wise order during parallel executions. Therefore, the global 32-bit *work_index* variable is split; the 16 most significant bits represent the row-wise index and the 16 least significant bits represent the column-wise index. This allows us to represent both indices using a single variable. One downside of this approach is the limitation on the total number of blocks that can be processed. The *work_size* cannot be larger than $(2^{16} - thread\_count)$ to prevent overflow into the other half of the variable, as the division between blocks processed in row-wise order and column-wise order is unknown. This limitation, however, can be resolved by changing the *work_index* to a 64-bit variable, but for the purpose of this thesis we decided to keep the size of the variable constant throughout the different algorithms.

The pseudocode for the two-sided block assignment loop, used in the assisting column-wise chained scan algorithm, is shown in Algorithm 2. The function header contains the following parameters:

- *work_size* - total number of blocks
- *work_index* - global block index variable
- *seg_count* - number of blocks used to process a single row
- *mult_rows_scan* - scan algorithm that combines multiple rows in single fixed size block
- *rw_scan* - row-wise chained scan algorithm
- *cw_scan* - column-wise chained scan algorithm

The if-statement on line 3, checks whether a single row can be contained within one fixed size data block, which is equal to the conditional (*row_size* <= BLOCK_SIZE) seen in Figure 6. If true, the *mult_rows_scan* algorithm is called, which is further explained in Section 3.4.3. Otherwise, we proceed to lines 10-16, in which several variables are initialised to be later used to synchronize the two-sided increment of the *work_index* variable. The assisting threads determine the progress made in row-wise order (lines 17-20), followed by an if-statement (lines 21-24) to decide whether they proceed in column-wise order (unclaimed row available) or assist the row-wise order scan.

**Algorithm 2** Two-sided parallel block assignment loop

```
1:  function TWO_SIDED_BLOCK_ASSIGNMENT(
        work_size, uint32* work_index, seg_count,
        mult_rows_scan, rw_scan, cw_scan)
2:      block_idx ← work_index
3:      if seg_count = 1 then
4:          /* Combine rows within a single block */
5:          loop
6:              mult_rows_scan(block_idx)
7:              block_idx ← ATOMIC_FETCH_ADD(
                    work_index, 1, ordering::relaxed)
8:              if block_idx >= work_size then break
9:      else
10:         rw_thread ← work_index = 0
11:         rw_idx ← block_idx ≫ 16
12:         cw_idx ← block_idx & 0xFFFF
13:         /* Synchronization variables */
14:         rw_claimed ← 0
15:         rw_work_size ← work_size
16:         cw_work_size ← 0
17:         if rw_thread = false then
18:             rw_claimed ← (rw_idx+seg_count−1) / seg_count
19:             rw_work_size ← rw_claimed * seg_count
20:             cw_work_size ←
                    work_size - rw_work_size
21:             if cw_work_size > 0 then
22:                 cw_scan(cw_idx, rw_claimed)
23:             else
24:                 rw_thread ← true
25:     loop
26:         if rw_thread then // Row-wise order
27:             res ← COMPARE_EXCHANGE_WEAK(
                    work_index, block_idx,
                    block_idx + 1 ≪ 16, ordering::
                    relaxed, ordering::relaxed)
28:             if res.is_ok() then rw_scan(rw_idx)
29:             block_idx ← LOAD(work_index,
                    ordering::relaxed)
30:             rw_idx ← block_idx ≫ 16
31:             cw_idx ← block_idx & 0xFFFF
32:             if cw_idx > 0 then
33:                 /* Determine synchroniza-
                       tion variables and set
                       rw_thread to false if cur-
                       rent row is finished */
34:             claimed ← rw_idx + cw_idx + 1
35:             if claimed > work_size then break
36:         else// Column-wise order
37:             block_idx ← ATOMIC_FETCH_ADD(
                    work_index, 1, ordering::relaxed)
38:             rw_idx ← block_idx ≫ 16
39:             cw_idx ← block_idx & 0xFFFF
40:             claimed ← rw_idx + cw_idx + 1
41:             if claimed > work_size then break
42:             if cw_idx ≥ cw_work_size then
43:                 rw_thread ← true
44:                 continue
45:             cw_scan(cw_idx, rw_claimed)
```

The two-sided loop is visible on lines 25-45, consisting of the row-wise order section (26-35) and the column-wise order section (37-45). When proceeding in row-wise order, we atomically increment the 16 most significant bits of *work_index* using the `compare\_exchange\_weak` function. On success, the *rw_scan* algorithm is called. Then we read the latest *work_index* value and calculate both indices (29-31), to determine whether parallel threads started assisting (line 32) or whether all blocks have been processed (lines 34-35). A similar pattern is used for the column-wise order. We atomically increment the 16 least significant bits of the *work_index* variable and determine both

indices (lines 37-39). Then we check whether all blocks have been claimed (lines 40-41) or whether just the column-wise order is finished such that the row-wise order scan can be assisted (lines 42-44). Otherwise, the *cw_scan* is called with the *rw_claimed* variable, representing the number of rows being processed in row-wise order.

### 3.4.3 Scan algorithm implementation

Now that all data blocks are being assigned to the working threads, we need to consider how to actually perform the scan over these blocks. As we've discussed in previous sections, the assisting column-wise chained scan requires knowledge about a few aspects: a location of the input and output arrays, information variables regarding the (multidimensional) shape of the data and an array of block descriptors. Therefore, each scan algorithm is first initialised using a `create_task` function. This function determines the necessary variables and arrays, which are combined in a single data structure to be send to the scan algorithm. Using this data structure and the incoming block index, the actual scan operation can be performed.

The pseudocode for the assisting column-wise chained scan algorithm is shown in Algorithm 3. The `create_task` function receives an input pointer, an output pointer and a shape, and creates a `Data` object (line 2) to be send to the `assisting_columnwise_scan` function. On line 5-7, we determine the size of the innermost dimension, the total number of rows and the number of blocks necessary to scan a single row. The if-statement on line 8, determines whether multiple blocks can be combined, which in turn effects the total number of block descriptors needed to be initialised in the descriptor array. These variables are combined into one data object (line 10), after which the assisting column-wise chained scan algorithm can be executed.

The `assisting_columnwise_scan` function first defines the different scan orders to be used by the block assignment loop, which is shown on lines 12-31. The `multiple_rows` function takes a block index as input parameter, and determines the number of rows that fit in a single block size. Using this information, the start and end variables of the current block are calculated, after which a sequential scan can be performed. The for-loop on line 18 is used to iterate over the different rows in the block. Once one block has been sequentially scanned, we increase the start index to the beginning of the next row, and repeat the same process until all rows are processed. The `rowwise_scan` function uses the block index to determine the current row and the offset on this row of the block to be processed (lines 22-23). Then, it calls the adaptive chained scan function to perform the scan over this block, as described in Section 2.2.1 and Section 2.2.2. The `columnwise_scan` function shows a similar pattern. However, it also determines the offset to be applied to the row index, to account for the number of rows that have already been processed in row-wise order. The algorithm concludes with a call to the two-sided block assignment loop (line 32) as described in Section 3.4.2, which decides when each of the three functions above should be applied.

**Algorithm 3** Assisting column-wise chained scan

```
1: const BLOCK_SIZE ← 4096
2: struct Data{T* input, T* output, blocks_per_row, inner_size, work_size,
                uint32 work_index, Descriptor* descriptors}
3: struct Descriptor{uint32 flag, T aggregate, T prefix}
4:     function CREATE_TASK(input, output, shape)
5:         inner_size ← GET_INNER(shape) // Retrieve the inner dimension
6:         inner_rows ← MULT_OUTER(shape) // Multiply dimensions, except the inner
7:         blocks_per_row ← (inner_size+BLOCK_SIZE−1) / BLOCK_SIZE
8:         block_count ← if blocks_per_row > 1 then blocks_per_row * inner_rows
                         else CEIL(inner_rows / (BLOCK_SIZE/inner_size))
9:         descriptors ← new Descriptor[block_count], init with X
10:        data ← Data{input, output, blocks_per_row, inner_size, block_count,
               work_index:0, descriptors}
11:        On available threads, execute ASSISTING_COLUMNWISE_SCAN(data)

12:    function ASSISTING_COLUMNWISE_SCAN(Data* d)
13:        inner_rows ← d→input.len() / d→inner_size
14:        function MULTIPLE_ROWS(block_index)
15:            rows_per_block ← MIN(BLOCK_SIZE / d→inner_size, inner_rows)
16:            size_block ← d→inner_size * rows_per_block
17:            start ← block_index * size_block
18:            for _ ← 0 to rows_per_block do
19:                SEQ_SCAN(&d→input[start], &d→output[start], d→inner_size, 0)
20:                start ← start + d→inner_size
21:        function ROWWISE_SCAN(block_index)
22:            row_idx ← block_index / d→blocks_per_row
23:            column_idx ← block_index % d→blocks_per_row
24:            descriptor_idx ← block_index // Block index in descriptor array
25:            ADAPTIVE_CHAINED_LOOKBACK(d, row_idx, column_idx, descriptor_idx)
26:        function COLUMNWISE_SCAN(block_index, completed_rows)
27:            new_inner_rows ← inner_rows − completed_rows
28:            row_idx ← (block_index % new_inner_rows) + completed_rows
29:            column_idx ← block_index / new_inner_rows
30:            descriptor_idx ← (row_idx * d→block_per_row) + column_idx
31:            ADAPTIVE_CHAINED_LOOKBACK(d, row_idx, column_idx, descriptor_idx)

32:        TWO_SIDED_BLOCK_ASSIGNMENT(d→work_size, &d→work_index, d→blocks_per_row,
                                      MULTIPLE_ROWS, ROWWISE_SCAN, COLUMNWISE_SCAN)
```

# 4 Experiments and Results

The performance of the different multidimensional parallel scan algorithms is measured using a variety of input shapes and sequences in both a prefix sum operation and an in-place prefix sum operation. In this context, the term 'in-place' indicates that the algorithm reuses the input array to store its computed result, which is achieved by referring both the input and output parameters of the algorithm to the same memory location. We compare the proposed assisting column-wise chained scan algorithm (Section 3.3) against the current multidimensional scan implementation of Accelerate (Section 2.3.1), the row-wise chained scan algorithm (Section 3.1) and the column-wise chained scan algorithm (Section 3.2). More specifically, the algorithms are tested against four different input shapes/matrices containing $n = 100.000.000$ 64-bit integers each, namely:

$$[[4, 25.000.000], [4.000, 25.000], [10.000, 10.000], [100.000, 1.000]]$$

These shapes are specifically selected to test the robustness of the algorithms in all kinds of multi-dimensional scenarios, including the edge cases as described in Section 2.3.1, a square $N$x$N$ matrix and a non-square $N$x$M$ matrix. Furthermore, a few benchmarks are included for one-dimensional sequences, higher dimensional shapes and the comparison between the row-wise and column-wise pattern, which will be further discussed in the next several subsections. The full implementation of these benchmarks can be found on Github[2].

We performed the benchmarks using an Intel 12900 processor, containing 24 threads divided among 8 performance cores and 8 efficiency cores. It has 80 KB L1 cache available for each core, runs two threads per core and offers a maximal memory bandwidth of 76.8 GB/s. Furthermore, it applies a *Uniform Memory Access* (UMA) architecture, meaning that each core has equal memory latency and access speed, therefore, eliminating slight performance differences caused by the access speed of the main core claiming the task. This is in contrast to *Non-Uniform Memory Access* (NUMA) architectures, for which the performance of a single run depends on the fact whether a core with low or high latency is performing the scan operation. Each benchmark first performs a cold run, in which no results are being recorded. Then the average execution times over 50 runs is calculated, to reduce the effect of variance in the final results. Although the processor contains a total of 24 threads, we only report the performance up to 16 threads, as applying more threads doesn't significantly affect the performance of the algorithm, neither positively nor negatively. Furthermore, the fixed block size used in the benchmarks is set to 4096 elements (64-bit/8-byte) as this fits within a single thread's L1 cache size (32 KB $<$ 40 KB).

## 4.1 Prefix sum

The performance of the multidimensional parallel scan algorithms is measured using the four input matrices as listed before. The corresponding results for the prefix sum and in-place prefix sum operations are presented in Figure 8 and Figure 9 respectively. The numeric representation of these

[2] https://github.com/Rudolf-UU/multi-dimensional-parallel-scan

graphs can be found in Table 2 and Table 3 in the appendix. Looking at these results, we can make the following observations.

The sequential row-based algorithm, currently used in Accelerate as described in Section 2.3.1, is proven to be a relatively simple yet effective method in most scenarios, apart from the tested edge case scenarios. For example, looking at Figure 8a, we can see that the performance of the sequential row-based algorithm no longer increases when applying more than 4 parallel threads, which is due to the limitation of only one thread being assigned to a single row in the input matrix. However, in the more general case scenarios, like the $N$x$N$ matrix, the algorithm is able to slightly outperform the other scan algorithms.

The assisting column-wise chained scan, is the only algorithm to show an increased performance with the small matrix input, shown in Figure 8d and Figure 9d. As a single row only contains 1000 elements and the applied fixed block size in the benchmarks is set to 4096 elements, the assisting column-wise chained scan algorithm is able to combine 4 rows of the input matrix into a single block, reducing the total number of atomic increments and memory calls.

The multidimensional parallel scan algorithms, apart from the column-wise chained scan, are able to perform single-threaded scan operations with zero overhead compared to the sequential scan. However, the performance of the column-wise chained scan on multi-threaded executions, especially up till 8 threads, is in most cases better than the row-wise chained scan. This difference between these two patterns is further explained in Section 4.2.

Lastly, we can see that all parallel algorithms reach their maximal performance around 6 to 8 threads, after which their performance slightly declines or remains the same. This is most likely caused by the limitation of memory bandwidth. As more threads join the computation, more blocks need to be fetched from memory to the local L1 caches. However, at a certain point the memory bandwidth will be fully utilised, meaning that other threads are idling until enough bandwidth is available to read their input values from memory or write their results back to memory. Based on these benchmark results, this point is reached around 8 parallel threads.

Referring back to the research questions, one goal of this thesis project is to answer the question: 'How to design a multidimensional parallel scan algorithm that can efficiently execute the scan operation over the available threads, regardless of the shape of the data structure?'. Looking at the discussed observations and the performance of the proposed assisting column-wise chained scan algorithm, we may conclude that the design of this algorithm forms a reasonable answer to that question. Throughout all different test scenarios, including higher dimensional benchmarks which are later discussed in Section 4.4, the performance of the assisting column-wise chained scan algorithm remains very similar. However, on the one hand we have the robustness of this algorithm in the edge case scenarios, while on the other hand we can see a slight performance loss in general case scenarios, compared to the performance of the sequential row-based algorithm. This forms a trade-off between the advantage and disadvantage of this algorithm, which has to be decided upon for each specific situation and application.

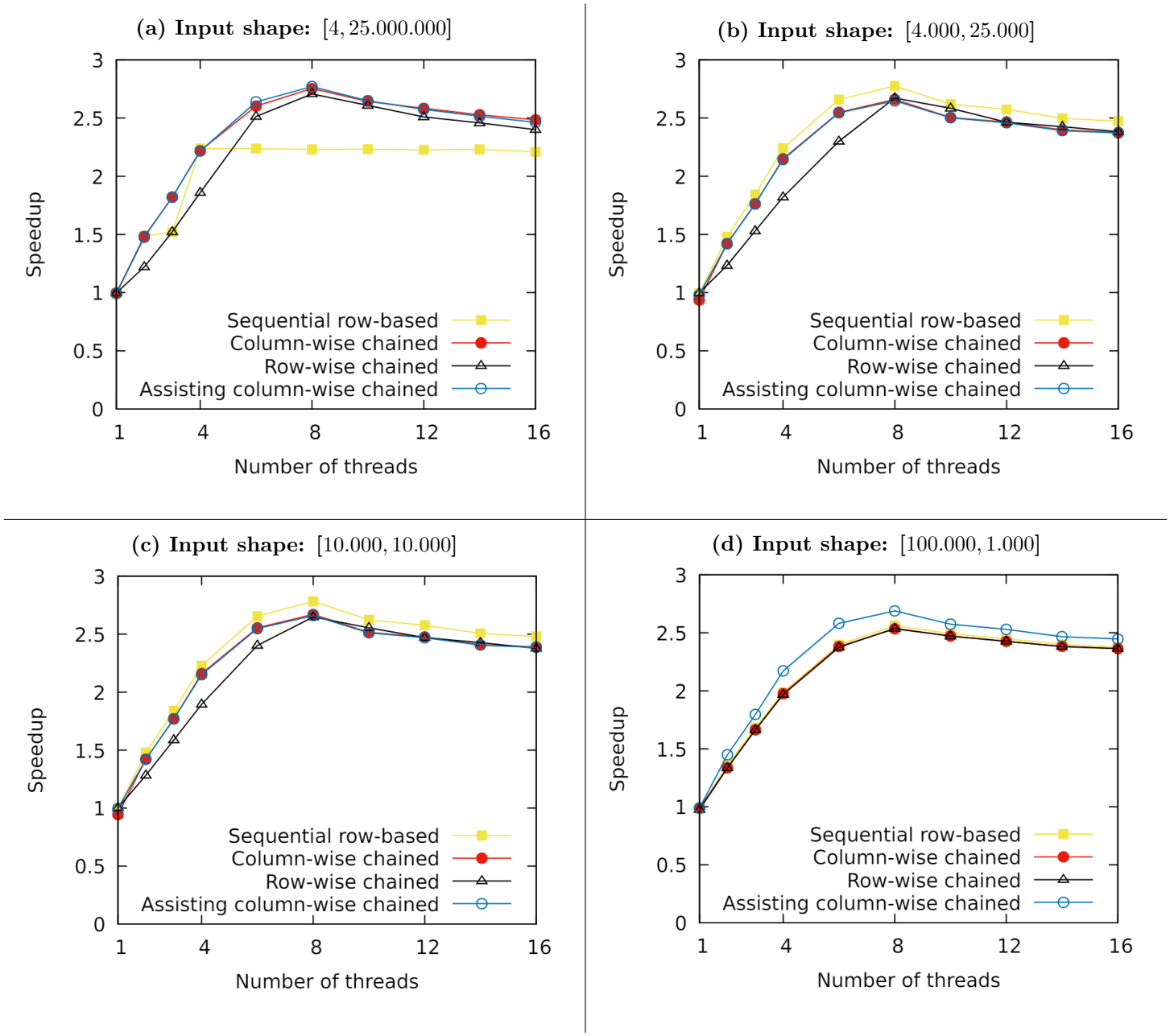



**Figure 8:** Prefix sum benchmarks of matrices with $n = 100.000.000$ elements. The results are normalised to the sequential scan performance.

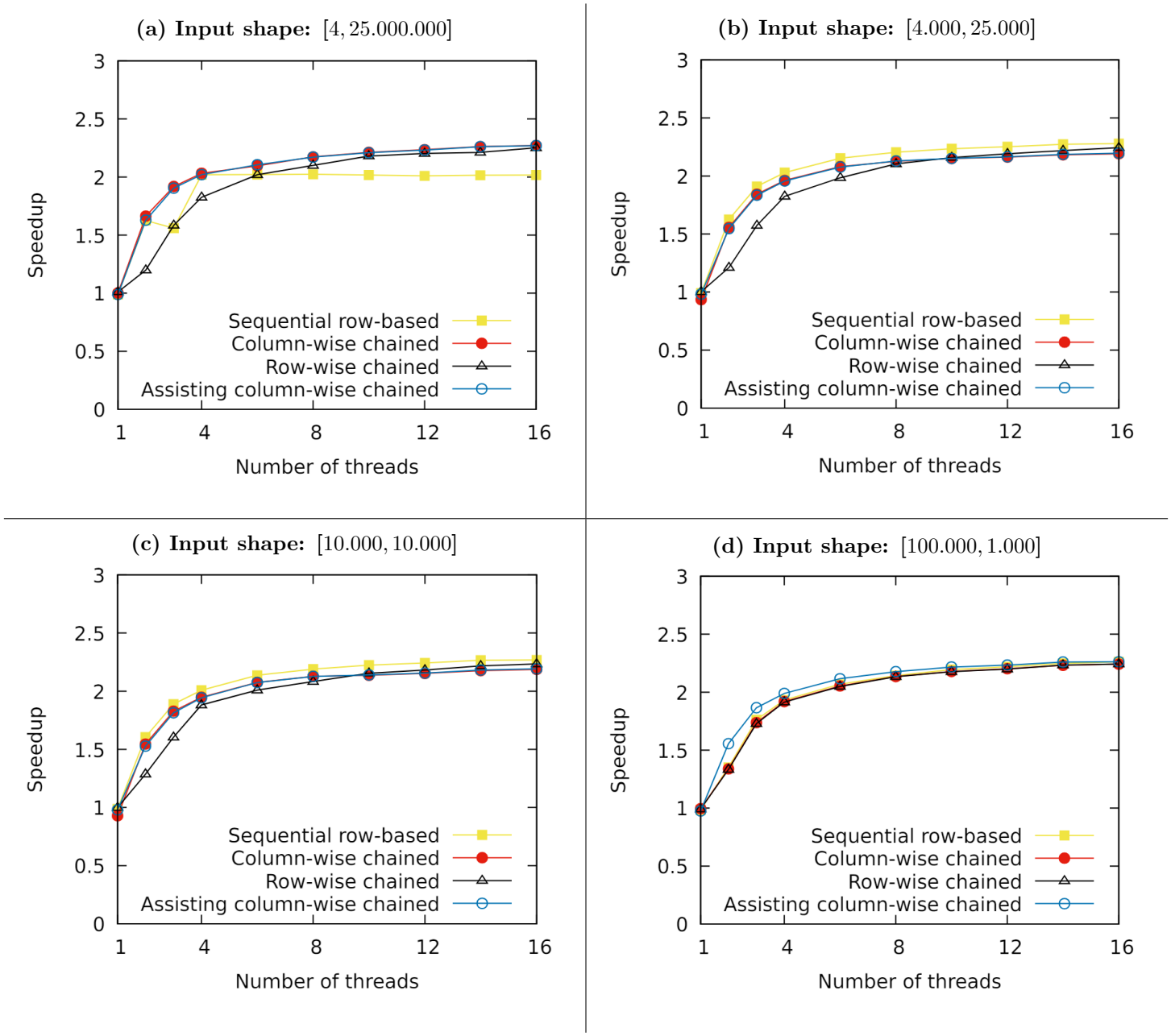



**Figure 9:** In-place prefix sum benchmarks of matrices with $n = 100.000.000$ elements. The results are normalised to the sequential scan performance.

### 4.2 Row-wise vs column-wise

To determine the best strategy for the assisting column-wise chained scan algorithm, as described in Section 3.3.1, we needed an overview of the strengths and weaknesses of both the row-wise and column-wise pattern. Table 1 presents the relevant rows of the prefix sum benchmark over the input matrix $[10.000, 10.000]$, visible in Figure 8c. This benchmark is used to show the relation between these two patterns/algorithms, but this relation is also recognisable in most other benchmarks. Based on these results, we can make a few observations.

First, the column-wise chained scan algorithm is most of the times performing equal or better than the row-wise chained scan algorithm. The most significant difference between the performance of these two patterns is visible between $2 - 8$ threads, while the difference between $8 - 16$ threads is instead relatively small. The first improvement is most likely achieved by the reduction of interference between consecutive threads, as described in Section 3.2. The small difference using 8 or more parallel threads is most likely caused by the limitation of memory bandwidth as described in the previous section.

Second, the row-wise chained scan algorithm performs single-threaded executions with no overhead compared to the sequential scan, while the column-wise chained scan has some overhead. We're unsure what the exact reason for the overhead is, however, a possible explanation could be related to the size of the L1 cache. The applied Intel 12900 processor, has per core 80 KB of L1 cache available and is able to run two threads per core, which means that 40 KB of L1 cache can be used by each thread. The fixed block size is based on this 40 KB, however, chances are that on a single-threaded execution a core is only launching a single thread. In that case, this single thread has 80 KB of L1 cache available, in which two fixed size blocks can be stored. As the row-wise pattern is processing the blocks in their original order, it is likely that the result of the previous block is still available in the cache, resulting in a low access time compared to a memory call. On the other hand, the column-wise pattern isn't processing direct consecutive blocks, meaning that the second result available in the cache is most likely irrelevant to the current block. Then, the result of the predecessor is fetched from memory, resulting in some overhead due to the higher access time. However, this explanation is purely based on reasoning, and further research is needed to get to a definitive answer.

| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
|---|---|---|---|---|---|---|---|---|---|---|
| Sequential | 1.00 | (67 ms) | | | | | | | | |
| Column-wise chained | 0.94 | 1.42 | 1.77 | 2.16 | 2.56 | 2.67 | 2.52 | 2.48 | 2.41 | 2.39 |
| Row-wise chained | 1.00 | 1.28 | 1.59 | 1.89 | 2.40 | 2.65 | 2.55 | 2.47 | 2.43 | 2.38 |

**Table 1:** Comparison of the row-wise pattern vs the column-wise pattern. Performance numbers are selected from the prefix sum benchmark using input [10.000, 10.000].

### 4.3 One-dimensional prefix sum

The assisting column-wise chained scan algorithm is based upon the concepts proposed in the adaptive chained scan algorithm (Section 2.2.2), which only supports one-dimensional input sequences. To measure any differences regarding the performance between these two algorithms, we applied them both to an one-dimensional input sequence. These benchmarks are equal to the prefix sum and in-place prefix sum benchmarks shown in the zero-overhead paper [6]. For simplicity, we excluded the proposed three-phase scans by that paper and instead directly compared the adaptive chained scan algorithm to the assisting column-wise chained scan algorithm. Figure 10 shows the result of these benchmarks over an array of $n = 2^{26}$ elements.

In these graphs is visible that both algorithms perform equally in the given one-dimensional scan operations. This means that the proposed assisting column-wise chained scan is not just able to handle multidimensional sequences, as seen in the previous sections, but is also able to efficiently scan one-dimensional sequences without any performance loss compared to the original adaptive chained scan algorithm. This is an useful property, from which frameworks like Accelerate could benefit. Usually these frameworks contain a separate function for the scan operation over one-dimensional and multidimensional sequences, as most scan algorithms can only handle one of these input variants efficiently. Using the assisting column-wise chained scan algorithm, however, both variants of input data can be handled by a single algorithm, simplifying the general structure of the framework and therefore making it less error prone to the programmer.

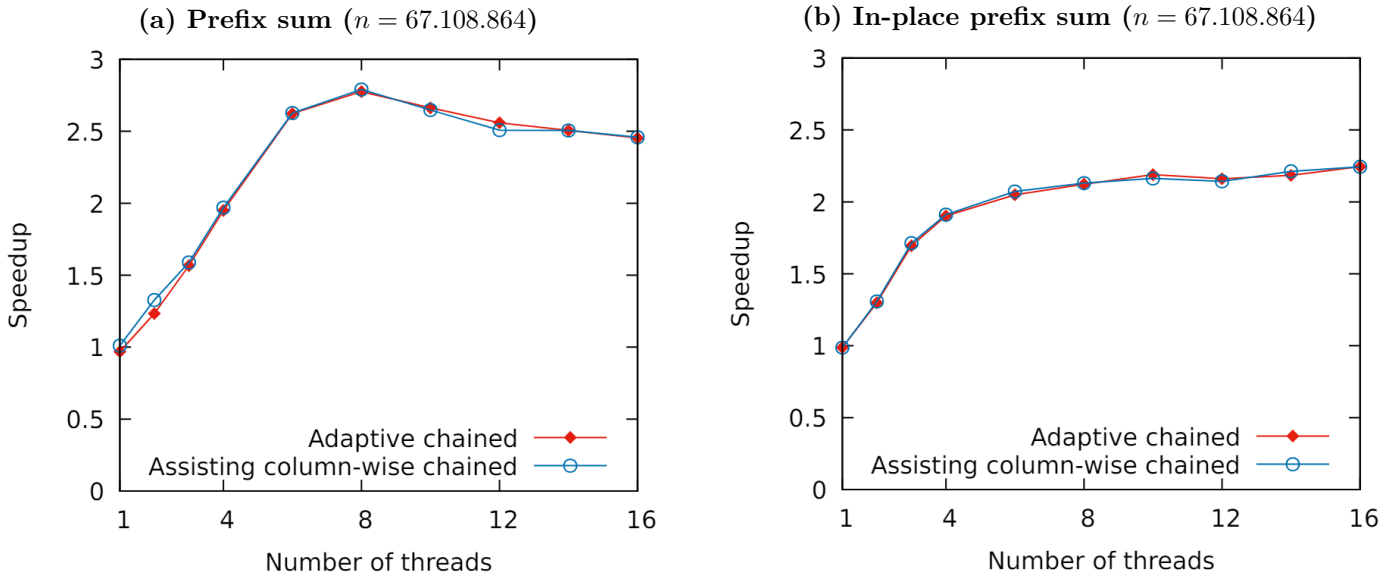



**Figure 10:** One-dimensional benchmark results, comparing the original adaptive chained scan against the proposed assisting column-wise chained scan.

### 4.4 Higher dimensional prefix sum

Benchmarks up until this point, only considered either one-dimensional sequences or matrices, which at itself doesn't prove an algorithm to be considered multidimensional. Therefore, we included some benchmarks that measure the performance of the different multidimensional scan algorithms over both a three-dimensional and four-dimensional input sequence, for which the results are visible in Figure 11. The performance of a multidimensional scan algorithm is expected to remain equal between two benchmarks, as long as the innermost dimension of the shape doesn't change, because only the innermost dimension is relevant for the scan performance. Therefore, we took the prefix sum benchmark over the shape $[10.000, 10.000]$, shown in Figure 8c, as our starting point. Then, we first split the shape over the outer dimension, resulting in the shape $[100, 100, 10.000]$, after which we split over the innermost dimension, resulting in the shape $[100, 100, 100, 100]$.

Comparing the benchmark result shown in Figure 11a with the original result in Figure 8c, it is visible that the performance of all algorithms indeed remained the same, as only the outer dimension has changed. However, the split over the innermost dimension resulted in some surprising results, which are visible in Figure 11b. The performance of the algorithms is expected to change as the innermost dimension is different, however, the decrease in performance during the two-threaded execution is surprising. The multidimensional scan algorithms, apart from the assisting column-wise chained scan, only outperformed the sequential scan when using at least six parallel threads. The reason for this decrease is somewhat unclear and would have to be further researched.

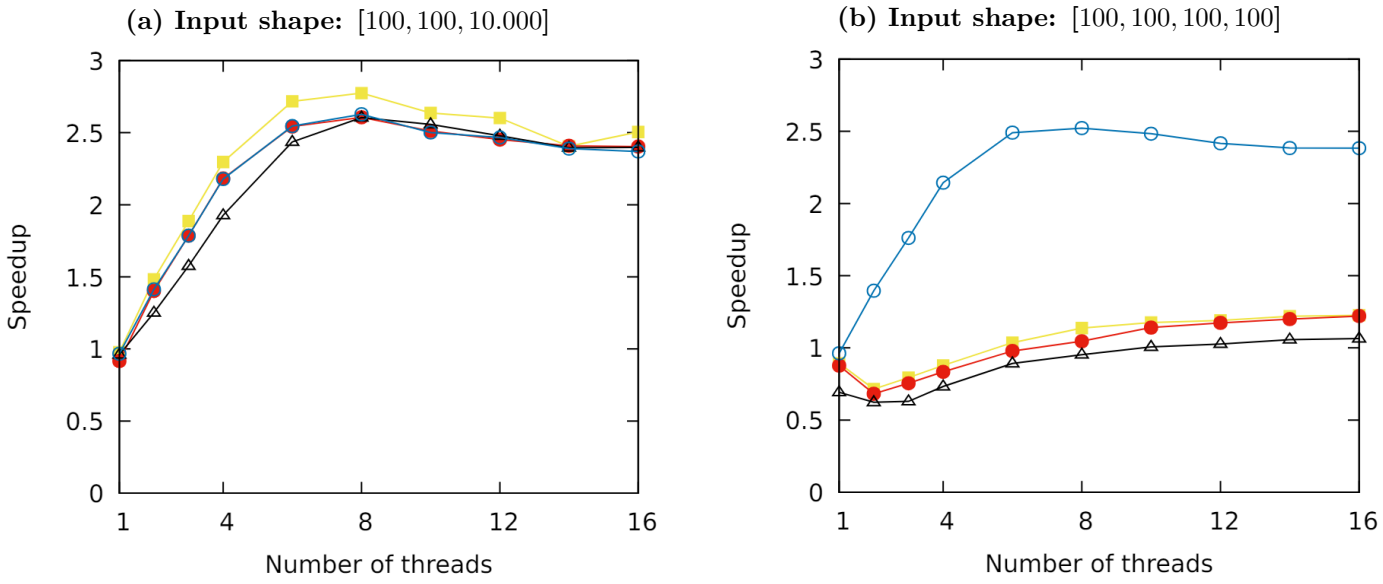



**Figure 11:** Higher dimensional prefix sum benchmarks, based upon the original $[10.000, 10.000]$ prefix sum benchmark (Figure 8c).

# 5 Related Work

The parallel execution of scans has been extensively studied for decades. As the capabilities of the *processing units* kept growing, this allowed researchers to constantly improve the performance of existing parallel scan algorithms.

Blelloch was one of the first to show the usability of the scan algorithm in parallel applications [2, 1], including radix sort and sparse matrix-vector multiplications [3]. These algorithms were designed for the CPU architecture, as the GPU was later introduced, which meant that the original algorithms were not directly interchangeable with the more complex architecture of the GPU. Horn [11] introduced one of the first GPU-based scan algorithm, which used the streaming model for the scan operation. Other researchers made use of the tree-based Brent-Kung scan [4], to develop the three-phase scan-then-propagate algorithm [16]. This was followed by the more efficient reduce-then-scan [7, 9], which reduces the number of required memory operations. Reducing the memory utilization at the cost of increasing the sequential dependencies between the threads, was the concept introduced with the chained scan algorithm [18, 15]. Fraser et al. [8] discussed the trade-off between accuracy and performance when it comes to sequences of floating point values, and introduces scan algorithms that reduce rounding errors.

While most of the algorithms were specifically designed for the GPU architecture, Zhang [19] proposed a novel parallel scan algorithm for multi-core processors. Previous work required the number of processors to be a power of two, while this algorithm could operate on any number of processors given at the start of the operation. The zero-overhead parallel scan paper [6], transformed efficient GPU scans to suitable version for multi-core CPUs. They reduced the overhead of these algorithms when executed on a single thread and allowed a variable number of threads to work on the elements during the execution.

Performing parallel scan operations over multidimensional data structures is a concept that has hardly been studied. As far as I'm aware, no research specifically targets this problem on the CPU architecture. Šinkarovs and Scholz [17] proposed a scan algorithm, that reshapes one-dimensional input sequences to multidimensional arrays, to allow the compiler to perform the sequential scan in parallel over the array shape. They allow the scan operation to be constructed in terms of smaller primitives. This, however, cannot be considered a general solution to the problem of performing a parallel scan over multidimensional input data, but instead is focused on the parallelization achieved by their specific compiler. Therefore, our research is an interesting addition to the current research upon parallel scan algorithms.

# 6 Conclusion

We introduced a parallel chained scan algorithm, capable of handling both one-dimensional and multidimensional data sequences. The one-dimensional performance matches the original *adaptive chained scan* algorithm [6], from which concepts are applied in the design of this algorithm. The proposed *assisting column-wise chained scan* starts by processing data blocks in row-wise order, and switches to column-wise order processing when parallel threads assist the computation. The row-wise order ensures zero overhead during single-threaded executions, while the column-wise order reduces interference between parallel threads during multi-threaded executions.

Benchmarks are used to compare our performance against the existing multidimensional scan in Accelerate [5] and the row-wise and column-wise order scans. Based on the results, we consider a trade-off between robustness and performance. The assisting column-wise chained scan is performing similar throughout all (multidimensional) test cases, including the considered *edge-case* scenarios of the existing Accelerate scan. Furthermore, both one-dimensional and multidimensional input sequences are efficiently being processed by this single algorithm. However, the necessary conditionals come at the cost of some overhead on '*non-edge-case*' input shapes, resulting in a slight performance loss against the existing algorithm in Accelerate. Therefore, depending on the specific application and/or framework, one algorithm may be preferred over the other.

Future work may focus on incorporating and evaluating this algorithm in an existing array-based language, like Accelerate, to compare the performance results in such a practical application. These languages often have other factors and optimisations, which is likely to have some effect on the performance of this algorithm. The property of handling both one-dimensional and multidimensional arrays, potentially simplifies the existing structure of languages, as many still apply a separate scan function for the one-dimensional and multidimensional input situation.

## References


[1] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38 (11):1526–1538, 1989.

[2] Guy E Blelloch. Prefix sums and their applications. 1990.

[3] Guy E Blelloch, Michael A Heroux, Marco Zagha, et al. *Segmented operations for sparse matrix computation on vector multiprocessors*. School of Computer Science, Carnegie Mellon University Pittsburgh, Pa, USA, 1993.

[4] Brent and Kung. A regular layout for parallel adders. *IEEE transactions on Computers*, 100 (3):260–264, 1982.

[5] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011.

[6] Ivo Gabe de Wolff, Gabriele Keller, David van Balen, and Trevor L McDonell. Zero-overhead parallel scans for multi-core cpus. To appear in PMAM 2024.

[7] Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213, 2008.

[8] Sean Fraser, Helen Xu, and Charles E Leiserson. Work-efficient parallel algorithms for accurate floating-point prefix sums. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2020.

[9] Sang-Won Ha and Tack-Don Han. A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(12): 2324–2333, 2012.

[10] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[11] Daniel Horn. Stream reduction operations for gpgpu applications. *Gpu gems*, 2(36):573–589, 2005.

[12] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[13] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *ACM SIGPLAN Notices*, 48(9):49–60, 2013.

[14] Trevor L McDonell, Manuel MT Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe runtime code generation: accelerate to llvm. *ACM SIGPLAN Notices*, 50(12):201–212, 2015.

[15] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

[16] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for gpu computing. 2007.

[17] Artjoms Šinkarovs and Sven-Bodo Scholz. Parallel scan as a multidimensional array problem. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–11, 2022.

[18] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–238, 2013.

[19] Nan Zhang. A novel parallel scan for multicore processors and its application in sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):397–404, 2011.

# A Numeric benchmark results

| **Prefix sum:** $[4, 25.000.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (66 ms) | | | | | | | | |
| Sequential (C++) | 1.03 | (65 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.49 | 1.52 | 2.24 | 2.24 | 2.23 | 2.23 | 2.23 | 2.23 | 2.21 |
| Column-wise chained | 1.00 | 1.48 | 1.82 | 2.22 | 2.60 | 2.75 | 2.64 | 2.58 | 2.53 | 2.48 |
| Row-wise chained | 1.00 | 1.22 | 1.52 | 1.86 | 2.51 | 2.71 | 2.61 | 2.51 | 2.46 | 2.40 |
| Assisting column-wise chained | 0.99 | 1.48 | 1.82 | 2.22 | 2.64 | 2.77 | 2.65 | 2.57 | 2.52 | 2.46 |

| **Prefix sum:** $[4.000, 25.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (66 ms) | | | | | | | | |
| Sequential (C++) | 1.03 | (65 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.48 | 1.84 | 2.24 | 2.66 | 2.78 | 2.62 | 2.57 | 2.50 | 2.47 |
| Column-wise chained | 0.94 | 1.42 | 1.76 | 2.15 | 2.55 | 2.66 | 2.50 | 2.47 | 2.40 | 2.38 |
| Row-wise chained | 1.00 | 1.23 | 1.53 | 1.82 | 2.30 | 2.67 | 2.58 | 2.46 | 2.43 | 2.38 |
| Assisting column-wise chained | 0.98 | 1.42 | 1.76 | 2.14 | 2.54 | 2.65 | 2.50 | 2.46 | 2.39 | 2.37 |

| **Prefix sum:** $[10.000, 10.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (67 ms) | | | | | | | | |
| Sequential (C++) | 1.03 | (65 ms) | | | | | | | | |
| Sequential row-based | 1.00 | 1.48 | 1.84 | 2.23 | 2.66 | 2.78 | 2.62 | 2.58 | 2.51 | 2.48 |
| Column-wise chained | 0.94 | 1.42 | 1.77 | 2.16 | 2.56 | 2.67 | 2.52 | 2.48 | 2.41 | 2.39 |
| Row-wise chained | 1.00 | 1.28 | 1.59 | 1.89 | 2.40 | 2.65 | 2.55 | 2.47 | 2.43 | 2.38 |
| Assisting column-wise chained | 1.00 | 1.42 | 1.77 | 2.15 | 2.55 | 2.66 | 2.51 | 2.47 | 2.41 | 2.39 |

| **Prefix sum:** $[100.000, 1.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (67 ms) | | | | | | | | |
| Sequential (C++) | 1.03 | (65 ms) | | | | | | | | |
| Sequential row-based | 0.98 | 1.36 | 1.68 | 1.99 | 2.40 | 2.56 | 2.49 | 2.45 | 2.40 | 2.37 |
| Column-wise chained | 0.99 | 1.34 | 1.66 | 1.98 | 2.38 | 2.53 | 2.47 | 2.43 | 2.38 | 2.36 |
| Row-wise chained | 0.98 | 1.34 | 1.66 | 1.97 | 2.37 | 2.54 | 2.47 | 2.43 | 2.38 | 2.36 |
| Assisting column-wise chained | 0.99 | 1.45 | 1.80 | 2.17 | 2.58 | 2.69 | 2.58 | 2.53 | 2.47 | 2.45 |

**Table 2:** Prefix sum benchmark results

| **In-place prefix sum:** $[4, 25.000.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (53 ms) | | | | | | | | |
| Sequential (C++) | 0.98 | (54 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.63 | 1.56 | 2.02 | 2.02 | 2.02 | 2.02 | 2.01 | 2.02 | 2.02 |
| Column-wise chained | 1.00 | 1.66 | 1.92 | 2.03 | 2.10 | 2.17 | 2.21 | 2.24 | 2.26 | 2.27 |
| Row-wise chained | 1.01 | 1.20 | 1.58 | 1.83 | 2.02 | 2.10 | 2.18 | 2.20 | 2.21 | 2.25 |
| Assisting column-wise chained | 0.99 | 1.63 | 1.90 | 2.02 | 2.11 | 2.17 | 2.21 | 2.23 | 2.26 | 2.27 |

| **In-place prefix sum:** $[4.000, 25.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (53 ms) | | | | | | | | |
| Sequential (C++) | 0.98 | (54 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.63 | 1.91 | 2.03 | 2.15 | 2.20 | 2.23 | 2.25 | 2.27 | 2.28 |
| Column-wise chained | 0.93 | 1.56 | 1.84 | 1.96 | 2.08 | 2.13 | 2.15 | 2.16 | 2.18 | 2.19 |
| Row-wise chained | 1.00 | 1.21 | 1.57 | 1.82 | 1.98 | 2.10 | 2.16 | 2.19 | 2.22 | 2.24 |
| Assisting column-wise chained | 0.98 | 1.55 | 1.83 | 1.96 | 2.08 | 2.13 | 2.15 | 2.16 | 2.19 | 2.20 |

| **In-place prefix sum:** $[10.000, 10.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (53 ms) | | | | | | | | |
| Sequential (C++) | 0.97 | (54 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.61 | 1.89 | 2.01 | 2.14 | 2.19 | 2.22 | 2.24 | 2.27 | 2.27 |
| Column-wise chained | 0.93 | 1.55 | 1.53 | 1.95 | 2.08 | 2.13 | 2.14 | 2.15 | 2.18 | 2.19 |
| Row-wise chained | 1.00 | 1.29 | 1.60 | 1.88 | 2.01 | 2.08 | 2.15 | 2.18 | 2.22 | 2.23 |
| Assisting column-wise chained | 0.98 | 1.53 | 1.81 | 1.94 | 2.07 | 2.13 | 2.14 | 2.16 | 2.18 | 2.19 |

| **In-place prefix sum:** $[100.000, 1.000]$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of threads** | **1** | **2** | **3** | **4** | **6** | **8** | **10** | **12** | **14** | **16** |
| Sequential (Rust) | 1.00 | (53 ms) | | | | | | | | |
| Sequential (C++) | 0.98 | (54 ms) | | | | | | | | |
| Sequential row-based | 0.99 | 1.35 | 1.76 | 1.93 | 2.07 | 2.14 | 2.20 | 2.22 | 2.25 | 2.26 |
| Column-wise chained | 0.99 | 1.34 | 1.74 | 1.92 | 2.05 | 2.13 | 2.18 | 2.20 | 2.23 | 2.24 |
| Row-wise chained | 0.99 | 1.33 | 1.73 | 1.91 | 2.05 | 2.13 | 2.18 | 2.20 | 2.23 | 2.24 |
| Assisting column-wise chained | 0.98 | 1.56 | 1.87 | 1.99 | 2.12 | 2.18 | 2.22 | 2.23 | 2.26 | 2.26 |

**Table 3:** In-place prefix sum benchmark results