Parallel Scan on Ascend AI Accelerators

Anonymous Authors[1]

Abstract

Large Language Models (LLMs) depend on model- and architecture-specific optimizations to be efficiently executed at a large scale. The number of new LLM variants grows rapidly, making it necessary to distill and only optimize a few common parallel computational primitives, including prefix sum (scan). In this work, we design high-performance prefix-sum algorithms for the Ascend NPU architecture and explore their applicability in AI workloads and LLMs. The key feature of our algorithms is the efficient use of vector and matrix units in the Ascend architecture, which allows us to reach up to 74.9% of the memory bandwidth achieved by memory copy. To showcase the effectiveness of matrix-multiplication-based scans as a fast primitive in AI workloads, we implemented several essential scan-based operators like radix sort and top-p, achieving respectively up to $3.3\times$ and $2.3\times$ speedup compared to the vector-only kernels. Finally, we show how these optimized kernels can impact real-world models on the Ascend NPU obtaining $2.02\times$ speed-up on state-space neural networks like Mamba, and up to $1.47\times$ on LLMs with large vocabulary sizes thanks to more efficient top-p token sampling.

An anonymized version of the source code is available at `https://gitlab.com/faceless-research/ascendc-scan`.

1. Introduction

Parallel scan is a fundamental parallel computing paradigm with many applications (Blelloch, 1990a; Lakshmivarahan & Dhall, 1994; Gu & Dao, 2024). Due to its importance, parallel scan has been studied in several models of computation, including the circuit (work and depth) and the Parallel Random-Access Machine (PRAM) models (Blelloch, 1996). In the circuit model, the depth and size tradeoffs for optimal parallel prefix circuits are well-understood for binary operations (Snir, 1986; Zhu et al., 2006), but in other models, especially in the heterogeneous computing domain (Dakkak et al., 2019; Zouzias & McColl, 2023) are under-explored.

Given the surge of AI, hardware vendors have manufactured heterogeneous AI accelerators with specialized compute engines for matrix multiplication known as *matrix engines* or *tensor core units*. A list of such specialized hardware units includes Google's TPUs (Jouppi & et al., 2017; 2021), Nvidia's Tensor Cores (NVIDIA Authors, 2017), AMD's Matrix Cores (AMD, 2022) and Huawei's Ascend Cube Unit (Liao et al., 2021; 2019) to name a few. See also (Rutledge, 2025) for a very recent example.

Although matrix multiplication still remains the dominant computational backbone in AI, the introduction of these matrix engines creates new opportunities for optimizing a broader range of computations. For this reason, a model of computation was recently proposed to capture matrix multiplication accelerators called the Tensor Core Unit (TCU) model (Chowdhury et al., 2020; 2021). The authors of (Chowdhury et al., 2021) initiated the study of TCU algorithms and revisited classical paradigms in the TCU model. The study of accelerating prefix sum (and reduction) operations using matrix multiplication units was first initiated in the seminal paper of (Dakkak et al., 2019). A follow-up work presented a logarithmic-depth parallel scan algorithm in the TCU model of computation, where only matrix multiplication operations are required (Zouzias & McColl, 2023). Although the main algorithm in (Zouzias & McColl, 2023) offers stronger theoretical guarantees than that of (Dakkak et al., 2019), its inefficient memory access patterns limit its practical performance. Here, we focus on new algorithms inspired by (Dakkak et al., 2019) that achieve performance close to hardware peak, and show their impact on AI workloads. Our top-performing multi-core scan kernels combine ideas from Alg. 6 of (Dakkak et al., 2019), explicit software-managed asynchronous pipelining between matrix and vector cores, and a novel matrix unit / vector unit recomputation strategy in the up-sweep scan phase. This strategy is different compared to all previously known

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

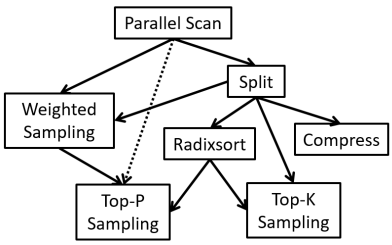

Figure 1. A diagram of well-known parallel scan applications considered here along with their dependencies.

scan strategies on accelerators (see Section 2.1), and not only provides concrete results for the Ascend architecture, but could be of interest to other accelerators as well. For example a follow-up study (Peng et al., 2025) builds upon the present work by introducing additional non-trivial techniques and reporting empirical improvements on GPU architectures.

As a case study and an evaluation environment, we use Huawei's Ascend AI accelerators and the Compute Architecture for Neural Networks (CANN) software ecosystem of Ascend to evaluate our proposals (Liang, 2020). It is important to emphasize here that *the vector cores of the current generation of Ascend NPUs are not designed to efficiently compute prefix sums*. Given the research outcomes here, optimizing prefix sums on the vector cores of Ascend without the utilization of the matrix cores might be unnecessary or a sub-optimal design strategy.

Our main contributions are listed below:

1. We designed and implemented parallel scan algorithms that make extensive use of the Ascend matrix multiplication engines ("Cube units"). In particular, we implement several scan variants, including single-core and multi-core scans, scans on multiple arrays (batched scan), inclusive/exclusive scans and specialized scan implementations for boolean (mask) inputs using the Cube units' 8-bit integer capabilities.

2. We implemented a list of computational kernels (operators), including parallel split, compress/compact, radix sort, top-k, and top-p (nucleus) sampling essential to AI workloads. In all of these cases, we demonstrate significant performance improvements against the vector-only and CPU implementations.

3. Last but not least, we show how these scan-based kernels directly impact the performance of state-space neural networks (SSNNs), taking as an example Mamba, and other Large Language Models (LLMs). The evaluation of these models was done with an out-of-the-box approach, meaning that we were able to improve performance by up to $2.02\times$ simply by replacing the PyTorch kernels under the hood without additional user code modifications. We avoided common optimization techniques like kernel-fusion, which can provide additional improvements but usually are model-specific. In any case, our algorithms are orthogonal to such optimizations.

2. Background

The inclusive prefix sum of a sequence of elements $x_1, x_2, x_3, \ldots$ equipped with a binary associative operator $\odot$ is the sequence $x_1, x_1 \odot x_2, x_1 \odot x_2 \odot x_3, \ldots$. Here, we focus on inclusive prefix sum, also referred to as $scan$, where the binary operator is addition.

2.1. Scan Strategies on Accelerators

Numerous parallel scan implementations and libraries have been proposed in the literature (Dotsenko et al., 2008; Yan et al., 2013; Hwu et al., 2023; Sengupta et al., 2007; Baxter, 2016; Bell & Hoberock, 2012; NVIDIA, 2023; Merrill & Garland, 2016). Horn was one of the first to implement parallel scans in GPUs (Pharr & Fernando, 2005), followed up with several improvements.

In a nutshell, GPU scan implementations primarily follow a two-level hierarchical approach where the highest level of the hierarchy is the block level. An efficient scan implementation follows one of these scan strategies: *Scan-Scan-Add (SSA)*, *Reduce-Scan-Scan (RSS)*, or *Stream-scan* according to the state-of-the-art *decouple lookback scan* approach of (Merrill & Garland, 2016). To the best of our knowledge, the state-of-the-art scan algorithm on GPUs follows a highly-tuned decoupled look-back and backoff approach, which is briefly explained in (Hemstad, 2024).

2.2. Applications of Parallel Scan

Parallel scan has a plethora of applications (Lakshmivarahan & Dhall, 1994; Blelloch, 1990a). Here, we restrict our attention to scan applications that enable us to generate efficient computational kernels (operators) that appear in AI workloads. In particular, we have identified that sorting, masking of tensors, and top-k/top-p sampling are essential. All these applications of the parallel scan are well-known, but the observation that top-p sampling can benefit from scan seems to be new (to the best of our knowledge).

In addition to these AI operators, (Smith et al., 2022) showed that structured SSMs (Gu et al., 2021) can also be run efficiently by parallelizing the sequential first-order recurrence with a parallel-scan. Recently, (Gu & Dao, 2024) proposed a new architecture called Mamba that incorporates selective SSMs, a variant of structured SSMs, into neural networks. Mamba can compete with transformers for language processing, and is especially efficient for long context lengths. A key requirement for this architecture is

to have an efficient and hardware-aware scan primitive.

3. Ascend AI Accelerators

In this section, we briefly discuss the DaVinci architecture of Ascend accelerators consisting of the cube and vector computing units, mostly following (Liao et al., 2019). We discuss the AscendC programming model, a recently proposed programming model for Ascend operator development in Appendix B. All the material presented here is available online at https://www.hiascend.com.

Huawei Ascend 910B is a recent series of Huawei chips designed to accelerate neural network training and inference. For the scope of the paper, an accelerator can be seen as a grid of computing units called AI Cores and a global High Bandwidth Memory (HBM) with L2 cache.

In the Ascend 910B series, an AI Core consists of one AI Cube (AIC) core and multiple, usually two, AI Vector (AIV) cores. Each AIC and AIV core contains a scalar unit for basic arithmetic operations, program flow control, calculating addresses, and dispatching instructions. Each core also includes computing engines (either vector or cube ones), local memory buffers, and Memory Transfer Engines (MTEs). MTEs are responsible for moving data between global and local memory buffers. Both MTEs and computing engines have separate instruction queues and work in parallel, so it is the programmer's responsibility to ensure synchronization.

An AI Vector core performs vector operations similar to traditional SIMD operations. The input and output data of an AI Vector core must be allocated to the local scratchpad/buffer called Unified Buffer (UB). AIV cores support simple arithmetic operations, such as vector addition as well as more complex ones (gather and reduce). The AI Vector core does not have hardware support for efficiently computing prefix sums.

An AI Cube core is primarily responsible for matrix multiplication operations. The AI Cube core contains a hierarchical scratchpad memory structure (L1, L0A, L0B, L0C, BT, FP buffers) and a cube computing engine. An AIC core can be configured to multiply two matrices of almost arbitrary sizes. The cube core supports both floating point and low-precision integer data types, i.e. input float16/float32 (with float32 output) and int8 (with int32 output).

Figure 2 shows the Ascend architecture where the Cube and Vector units are separate cores. In the 910B architecture, data can only be exchanged using global memory and/or L2 cache.

4. Scan Algorithms on Ascend

We discuss the design and implementation of parallel scan algorithms using matrix multiplication accelerators. Although the Ascend AI accelerator is used as a case study for the implementations, we aim to decouple the fundamental algorithmic ideas from the intricate architectural details of the Ascend accelerator as much as possible.

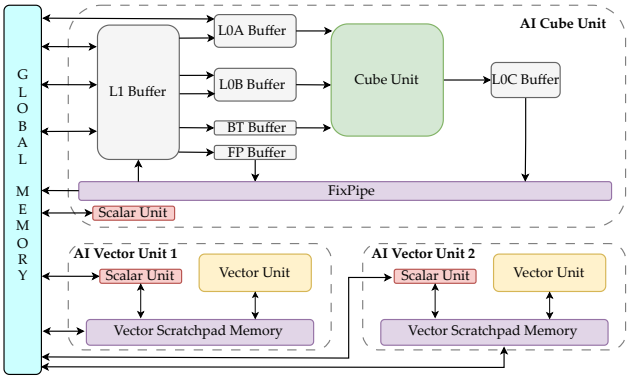

Figure 2. Architecture of Ascend 910B accelerators. Each AI core contains one cube and two vector units.

Matrix multiplication is an essential operator in our discussion here; hence, we introduce some linear algebraic notation that will be used throughout the paper. We denote matrices using boldface font and capitals, i.e., $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$. We use s to denote the size of square matrices, i.e., $\boldsymbol{A}_s$. We drop the subscript on the matrices when the dimension is clear from the context. We denote matrix multiplication between $\boldsymbol{A}$ and $\boldsymbol{B}$ by $\boldsymbol{C} := \boldsymbol{A} \mathbin{@} \boldsymbol{B}$. We denote by $\boldsymbol{U}_s$ the upper-triangular all-ones square matrix of size s, including ones on the main diagonal (zeros otherwise). $\boldsymbol{L}_s$ is the lower triangular all-ones of size s. $\boldsymbol{L}_s^-$ is the *strictly* lower triangular all-ones of size s: $\boldsymbol{L}_s^-$ has zeroes on the main diagonal. $\mathbf{1}_s$ denotes the all-ones square matrix of size s. We denote one-dimensional arrays using boldface font and lower-case: $\boldsymbol{r}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$. We frequently partition an array $\boldsymbol{x}$ into tiles of length s^2, i.e., tiles are contiguous blocks of s^2 entries of $\boldsymbol{x}$. We note an arbitrary s^2-tile of $\boldsymbol{x}$ by $\boldsymbol{x}_{s^2}$. We also view a tile $\boldsymbol{x}_{s^2}$ as a row-major matrix $\boldsymbol{A} = \text{view}(\boldsymbol{x}_{s^2})$ having s rows and s columns (zero-pad if necessary).

In Section 4.1, we present two scan algorithms (Algorithm 4.1 and Algorithm 4.2) that use a single matrix multiplication unit; we call these algorithms respectively `ScanU` and `ScanUL1`, based on which constant matrices they use. The key ingredient here is to utilize Ascend's cube unit effectively. In particular, Algorithm 4.2 performs multiple matrix multiplications and utilizes the accumulation buffer of the Cube unit to compute the scan of an input tile of length s^2, whereas Algorithm 4.1 computes s consecutive scans of smaller tiles of length s using a single matrix multiplication.

In Section 4.2, we present a multi-core scan algorithm (MCScan, Algorithm 4.3) tailored for the Ascend AI accelerator. A key feature of MCScan is that it utilizes all the available cube and vector cores. MCScan and its extensions are designed for scenarios involving very large one-dimensional arrays.

4.1. Warm-up: single cube scans

Here, we present two scan algorithms that utilize a single cube and vector units. Both algorithms are tailored to the DaVinci architecture and are based on the linear algebra fact that if $\boldsymbol{A}$ is the row-major matrix view with s columns of a vector $\boldsymbol{x}$ then:

> Matrix multiplication $\boldsymbol{A} \mathbin{@} \boldsymbol{U}_s$ computes "local" scans of tiles of size s of $\boldsymbol{x}$.

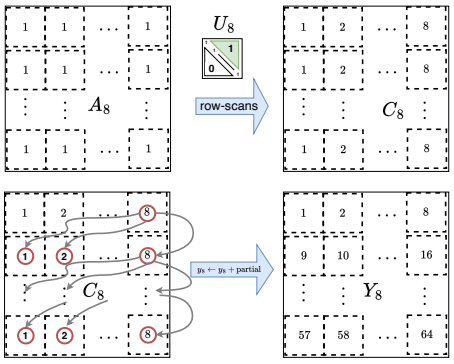

Figure 3. Example of a single tile iteration of length $8^2 = 64$ of ScanU (Algorithm 4.1). Input $\boldsymbol{A}_8$ is all-ones, whereas output $\boldsymbol{Y}_8$ is the sequence $1, 2, \ldots, 64$.

The first algorithm, ScanU (Algorithm 4.1), computes s consecutive local scans of tiles of size s using the cube unit and then propagates the partial sums using a single vector unit. Figure 3 depicts a single (tile) iteration of Algorithm 4.1 with $s = 8$. The input matrix tile $\boldsymbol{A}_8 = \text{view}(\boldsymbol{x}_{64})$ used as an example is the all-ones matrix, and the expected output is $1, 2, \ldots, 64$. In the first step, $\boldsymbol{C}_8 := \boldsymbol{A}_8 \mathbin{@} \boldsymbol{U}_8$ computes the row-scans and in the second step the last entry of each row of $\boldsymbol{C}_8$ is broadcast-added to its next full-row.

The second algorithm, ScanUL1 (Algorithm 4.2), is an Ascend adaptation of Algorithm 6 of (Dakkak et al., 2019) and is based on a matrix identity that expresses the scan of an array $\boldsymbol{z}$ of length s^2 using matrix operations. View $\boldsymbol{z}$ as a square row-major matrix $\boldsymbol{A}_s = \text{view}(\boldsymbol{z})$ of size $s \times s$ (pad with zeros if needed). Given $\boldsymbol{z}$, the inclusive scan of $\boldsymbol{z}$ ($\texttt{scan}(\boldsymbol{z})$) can be computed as:

$$\texttt{scan}(\boldsymbol{z}) = \boldsymbol{A}_s \mathbin{@} \boldsymbol{U}_s + \boldsymbol{L}_s^- \mathbin{@} \boldsymbol{A}_s \mathbin{@} \boldsymbol{1}_s, \qquad (1)$$

ignoring any padded values. Equation 1 first appeared in (Dakkak et al., 2019). ScanUL1 uses Equation 1 to scan each consecutive tile of size s^2 (Lines 6-12 of Algorithm 4.2), and then propagates the last value of the partial sums sequentially. In a high-level, for each tile of size s^2, the cube unit evaluates Equation 1 with the following sequence of matrix operations:

$$\begin{aligned}\boldsymbol{C}_1 &= \boldsymbol{A}_s \mathbin{@} \boldsymbol{1}_s \\ \boldsymbol{C}_2 &= \boldsymbol{A}_s \mathbin{@} \boldsymbol{U}_s \\ \boldsymbol{C}_2 &= \boldsymbol{C}_2 + \boldsymbol{L}_s^- \mathbin{@} \boldsymbol{C}_1.\end{aligned}$$

The above sequence of matrix operations has two advantageous properties with respect to data movements. The first two steps of the above sequence share the left matrix operand $\boldsymbol{A}$, allowing us to load $\boldsymbol{A}$ only once in L0A. Moreover, the third step effectively utilizes the accumulation buffer of the cube unit since $\boldsymbol{C}_2$ is reused in the last two steps. Once the local scan of a tile of size s^2 is computed, a single vector core adds the last value of the previous scanned tile to the current tile (see Lines $14 - 16$ of Algorithm 4.2).

Algorithm 4.1 Scan Cube-Vector (ScanU)

```
1:  procedure ScanU(x, s)
2:      partial ← 0                          ▷ Accumulation value (Vector Unit)
3:      Load U_s in L0B
4:      for each s^2-tile of x: x_{s^2} do   ▷ Pipelined exec.
5:          Load x_{s^2} to L0A              ▷ Cube unit
6:          C ← A_s @ U_s                    ▷ acc. off, free L0A
7:          Copy C from L0C to y_{s^2} in GM
8:          Vector unit waits tile from Cube unit
9:          Copy y_{s^2} from GM to UB       ▷ Vector unit
10:         for each s-tile of y_{s^2}: y_s do
11:             y_s ← y_s + partial
12:             partial ← last entry of y_s
13:         Copy y_{s^2} from UB to GM
14:     Return y
```

Algorithm 4.2 ScanUL1 is adaptation of Alg. 6 (Dakkak et al., 2019)

```
1:  procedure ScanUL1(x, s)
2:      partial ← 0                          ▷ Accumulation value (Vector Unit)
3:      Load U_s, L_s^-, 1_s in L1
4:      for each s^2-tile of x: x_{s^2} do   ▷ Pipelined exec.
5:          Load x_{s^2} to L0A and 1_s to L0B   ▷ Cube unit
6:          C_1 ← A_s @ 1_s
7:          Copy C_1 from L0C to L1
8:          Load U_s to L0B
9:          C_2 ← A_s @ U_s
10:         Load L_s^- in L0A and C_1 in L0B
11:         C_2 ← C_2 + L_s^- @ C_1
12:         Copy C_2 from L0C to y_{s^2} in GM
13:         Vector unit waits tile from Cube unit
14:         Copy y_{s^2} from GM to UB       ▷ Vector unit
15:         y_{s^2} ← y_{s^2} + partial
16:         partial ← last entry of y_{s^2}
17:         Copy y_{s^2} from UB to GM
18:     Return y
```

Both algorithms process their tiles using a pipelined execution strategy. The critical path or span of both proposed algorithms is linear on the input length for constant values of s, since there is a sequential dependency on the partial sums. Therefore, these kernels are more effective when the input array to be scanned has a relatively short length. Additionally, designing and implementing scan algorithms that use a single cube core is a building block for extending these ideas to a multi-core scenario, as seen in Sections D.1 and 4.2.

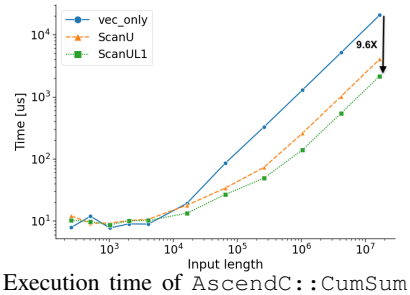

Figure 4. Execution time of `AscendC::CumSum` (vec_only) versus `ScanU` and `ScanUL1` (log-log scale).

Does cube utilization imply performance? Although the above scan algorithms demonstrate that it is possible to utilize the cube units for scan, it is unclear if cube utilization translates to performance improvements compared to "vector only" scan algorithms. We provide an experimental evaluation demonstrating the benefits of using the Cube unit. We developed a vector-only kernel that uses the `CumSum` AscendC API[1] Figure 4 demonstrates a significant performance improvement (5× for `ScanU`, and 9.6× for `ScanUL1`) compared to the vector-only `CumSum` algorithm.

4.2. Multi-core Scan

This section presents a multi-core scan algorithm (`MCScan`, Algorithm 4.3) to compute the prefix sum of large input arrays. For the sake of clarity, the presented algorithm employs a 1-to-1 vector-to-cube core ratio. Our implementation takes advantage of the 2-to-1 ratio of 910B, but we consider it an implementation detail.

`MCScan` is similar to the Scan-Scan-Add (SSA) paradigm of computing scans using a hierarchical partition of the input into blocks (top-level of the hierarchy) and tiles within each block, as discussed in Section 2.1. `MCScan` consists of two phases separated by a global synchronization barrier across all cores (blocks). In contrast to previous methods, `MCScan` performs parallel partial re-computation of the block-level reductions during its first phase, as we will explain next.

[1]CumSum API documentation available at https://www.hiascend.com (accessed on 25 August 2024).

In the first phase, the cube and vector units work in parallel to partially compute the first scan part of SSA simultaneously. The cube units compute the local prefix sums of all consecutive (row) tiles of size s and write them back to global memory. In parallel with the cube units, the vector units compute the reduction over the tiles and then hierarchically reduce the tile reductions on a block-level granularity. The result of these reductions is written in an array $\boldsymbol{r}$ where the i-th entry equals the reduction of all the values of the i-th block. By definition, $\boldsymbol{r}$ has length equal to the number of blocks, B.

In the second phase, the vector units read the local s-tile scans and reduction values per block $\boldsymbol{r}$ from global memory. First, every vector core independently computes the prefix sum of the array $\boldsymbol{r}$ in its local scratchpad memory `UB`, i.e., performs a "small" scan on the block-level reduction. Next, each vector core uses the scanned reduction values to propagate (add) the results of the local s-tiled scans.

Algorithm 4.3 Multi-core Scan (`MCScan`)

procedure MCSCAN($\boldsymbol{x}$, s, B) ▷ B: number of blocks
  parfor i-th block of $\boldsymbol{x}$: $\boldsymbol{x}[i]$ **do** ▷ Phase I
    Load $\boldsymbol{U}_s$ in `L0B` ▷ Cube Units
    for each s^2-tile of $\boldsymbol{x}[i]$: $\boldsymbol{x}_{s^2}$ **do**
      Load $\boldsymbol{x}_{s^2}$ from `GM` to `L0A`
      $\boldsymbol{C} \leftarrow \boldsymbol{A}_s \mathbin{@} \boldsymbol{U}_s$
      Copy $\boldsymbol{C}$ in $\boldsymbol{y}[i]$ in `GM`
    Load $\boldsymbol{x}[i]$ to `UB` ▷ Vector Units
    $r_i \leftarrow$ REDUCESUM($\boldsymbol{x}[i]$)
    Write r_i on i-th entry of $\boldsymbol{r}$ in `GM`
  end parfor
  SyncAll: Synchronize all cube/vector cores
  parfor i-th block of $\boldsymbol{y}$: $\boldsymbol{y}[i]$ **do** ▷ Phase II
    Load $\boldsymbol{r}$ from `GM` to `UB`
    partial $\leftarrow$ Sum first i entries of $\boldsymbol{r}$
    for each s^2-tile of $\boldsymbol{y}[i]$: $\boldsymbol{y}_{s^2}$ **do**
      for each s-tile of $\boldsymbol{y}_{s^2}$: $\boldsymbol{y}_s$ **do**
        $\boldsymbol{y}_s \leftarrow \boldsymbol{y}_s +$ *partial*
        partial $\leftarrow$ last entry of $\boldsymbol{y}_s$
      Copy $\boldsymbol{y}_{s^2}$ from `UB` to `GM`
  end parfor
  Return $\boldsymbol{y}$

5. Operators based on Scan

In this section, we revisit several computational parallel primitives based on parallel scans (Blelloch, 1989; Sengupta et al., 2007) which are fundamental in AI workloads.

Interestingly, current AI workloads like Large Language Model (LLM) inference make implicitly heavy use of scan-based computational primitives, including top-k and top-p (nucleus) sampling (Holtzman et al., 2020), see also (Gu & Dao, 2024). The top-p sampling implementation of the popular open-source model Llama3 (Touvron et al., 2023) contains a batched sorting and prefix sum operation as the

first two PyTorch operations, see (Meta AI, 2024).

The scan-based operations that we consider are the following:

- `Split`: given an input array and a boolean flag array, split places all the elements where the corresponding flag is true at the beginning of the output array, followed by all items where the corresponding flag is false. `Split` executes an exclusive scan using `MCScan` on the mask array. Afterwards, it gathers the correct input elements and their indices, using vector core's `GatherMask` instruction and it stores them in global memory at the offsets calculated by the scan.
- `Compress`: Compress is a particular case of split in which only the first part of the output elements of the split are returned. Compress is equivalent to the PyTorch `torch.masked_select` operator.
- `Radix-sort`: Radix sort is a well-known application of split (Blelloch, 1990a; Blelloch et al., 1991). We implement a Least-Significant Bit (LSB) radix sort in AscendC using the split operator based on the `MCScan` algorithm. We implemented an additional vector-only kernel, `RadixSingle`, that extracts the radices of the inputs before the execution of the split. `RadixSingle` makes use of the AscendC vector instructions `ShiftRight` and `Not` to create the input mask for split. Additional pre-processing and post-processing phases are needed to support floats; see Exercises 8 and 9 in Section 5.2.5 of (Knuth, 1998).
- `Top-`k: Top-k selection is an essential operation in Large Language Models (LLMs) inference where the output tokens are typically sampled from the k tokens having the highest probability for the given context (Kool et al., 2019). Top-k has been recently used also in the Deepseek v3.2 model to implement the Lightning Indexer (DeepSeek-AI, 2025). The interested reader is referred to a recent survey on parallel top-k (Zhang et al., 2023). We implemented top-k on top of split using a partial quick-sort/select approach (Blelloch, 1996)
- `Top-`p: Top-p sampling in Large Language Model inference is an operation that applies sort and scan on the token probability vector (Holtzman et al., 2020). Interestingly, if the sorting step is implemented using radix sort, the top-p sampling operator becomes a scan-intensive operator!

6. Experimental Evaluation

In this section, we evaluate the multi-core scan, radix sort, top-k, and top-p sampling algorithms presented in the previous sections using Ascend AI Accelerators. The evaluation of other algorithms like batch scan and compress can be found in the Appendix D.

We have implemented all proposed algorithms in C++17 using the AscendC programming framework detailed in Appendix B. We used the Ascend CANN toolkit 8.2.rc1 with Ascend firmware and drivers versions 1.0 and 23.0.0, respectively. All evaluations are performed on Huawei's Ascend 910B4 accelerator. In particular, 910B4 contains 20 Cube Units and 40 Vector Units (the vector-to-cube units ratio is 2-to-1). The theoretical memory bandwidth of 910B4 is 800 GB/s. The host CPU is an AMD EPYC 9654 96-Core Processor having a theoretical peak memory bandwidth of 460.8 GB/s.

We compare against many-core CPUs using ParlayLib: A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines (Blelloch et al., 2020). In addition, we compare the proposed scan operators with H200 GPUs (CUDA 12.8) with a theoretical memory bandwidth of 4.8 TB/s.

6.1. Multi-core Scan (`MCScan`)

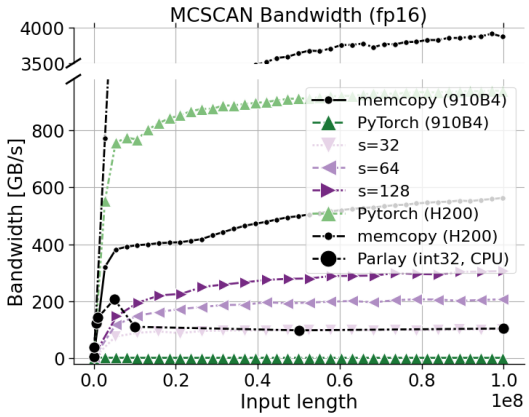

Figure 5. Bandwidth of `MCScan` (Algorithm 4.3). `MCScan` has 15.2× speedup against `ScanU` on 910B4 (20 AI cores).

Figure 5 depicts the performance of `MCScan` (Algorithm 4.3) on Ascend 910B4 versus the state-of-the-art (`torch.cumsum`). We integrated the multi-core algorithm in PyTorch to ensure a fair comparison and expose it as a custom PyTorch operator. The baseline operator doesn't use the cube unit, while `MCScan` takes advantage of all the computing units. We compare `MCScan` to a `memcopy` kernel that performs a memory copy. In this comparison, we reach 54.3% of the `copy`'s bandwidth (see Figure 6 for further improvements), while on H200 `torch.cumsum` can reach only 24.0% of the corresponding memory copy bandwidth.

A clear trend is that the larger the matrix multiplication size s is, the better the performance of the `MCScan`. $s = 128$

(almost) maximizes the utilization of the level-0 scratchpad memories L0A and L0B of the cube unit. We foresee that increasing the matrix multiplication size could lead to further performance improvements.

6.1.1. MCScan Optimizations

We have implemented two optimizations for MCScan. First, we implemented an L2 cache optimization that we call "L2 cache splitting". This optimization splits the input into consecutive chunks so that the input and intermediate arrays fit in the L2 cache during the processing of each chunk. Each chunk is processed serially, and the last value of each chunk is propagated to the next chunk to carry ahead the partial prefix sums. This optimization, integrated in the MCScan, shows a performance improvement of 25%.

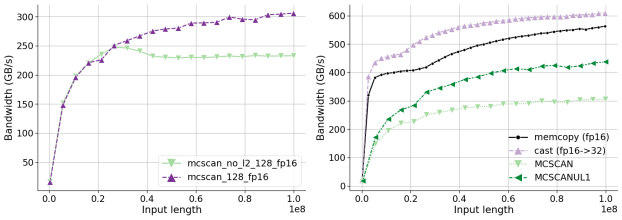

Figure 6. Improvements of MCScan using L2 cache splitting (left) and ScanUL1 (right).

Second, we designed and implemented a multi-core ScanUL1 algorithm for fp16 by replacing the single matrix multiplication during Phase I of MCScan with the three matrix multiplications of ScanUL1. We call the resulting algorithm, MCScanUL1. Figure 6 shows a performance improvement of 33% of MCScanUL1 versus MCScan and MCScanUL1 reaches up to 74.9% of the memcopy memory bandwidth as advertised in the abstract. Figure 14 of (Dakkak et al., 2019) reports around 70-75 Gelems/s on V100 GPU which is comparable versus 70.57 Gelems/s performance of MCScanUL1.

6.2. Radix Sort

We modified the radix sort operator to additionally return indices that correspond to the input index of each output element. This modification ensures a fair comparison with the sort operator provided by the PyTorch Ascend adapter. Our radix sort implementation is stable and supports unsigned (or signed) integers and floats (fp16).

Figure 7 shows the performance of a parallel fp16 radix sort implementation using MCScan with input data type int8 (Algorithm 4.3) to perform the parallel split step. For input lengths greater than 525K, our implementation of radix sort delivers a speedup between 1.3× and 3.3× compared to the `torch.sort()` baseline. Parlay is the CPU performance of the benchmark `integer_sort_pair<int16_t>`

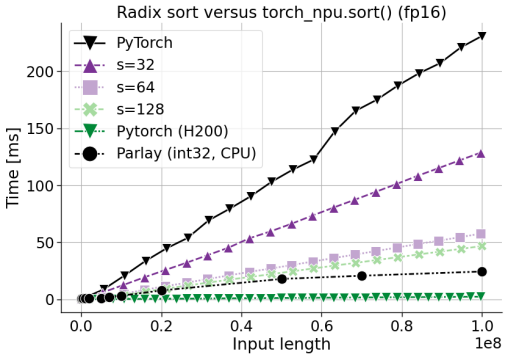

Figure 7. Radix sort versus baseline `torch_npu.sort` (fp16).

which is ≈ 3× faster than our best performing ($s = 128$). Additional performance is to be expected for low-precision data types (low bit-width) since the number of radix sort iterations equals the input bit-width. Indeed, the trend in AI accelerators is to introduce low-precision formats (NVIDIA Corporation, 2020), therefore, we expect a 2× improvement for radix sorting in low-precision 8-bit scenarios without any extra development effort, see Figure 15.

6.3. Top-k

Figure 8 shows the execution time of Top-k ($k = 2048$) as in the Lightning Indexer but with input data type int16 (fp8 is not supported); see (DeepSeek-AI, 2025). In this implementation, we use a naive pivot selection where the pivot is the average between the maximum and minimum of the input array. For fp16, our implementation reaches 80% of the baseline performance, but we believe that a better pivot selection strategy could drastically improve the algorithm. We reserve such improvements for future work.

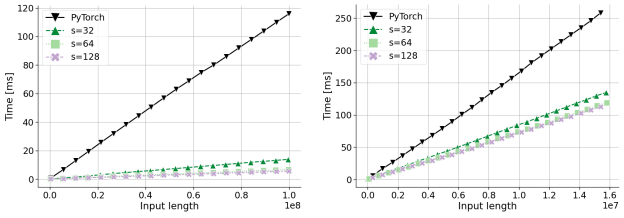

Figure 8. Execution time in milliseconds for top-k (left) and top-p sampling (right). The baseline is labeled PyTorch.

6.4. Top-p Sampling

Figure 8 depicts the execution time of drawing one sample using the top-p sampler as it is implemented in the Llama3 model, see (Meta AI, 2024). PyTorch corresponds to the Ascend implementation using the baseline sort and cumsum operators. The lineplots with labels $s = 32, 64, 128$ are similar to the baseline by replacing the `sort()` and `cumsum()` operators with the proposed radix sort and

multi-core scan, respectively.

7. The impact of scan on LLMs

Given the scan-based kernels we presented above, we now evaluate their end-to-end performance impact on some of the most common Large Language models (LLMs). Here we strive to improve performance for as many models as possible using a single optimized family of scan-based kernels. Therefore our evaluation on LLMs compares the standard PyTorch implementations with a version in which we replaced the underlying kernels of several PyTorch operators (i.e., `torch.sort` and `torch.cumsum`) with optimized kernels.

The goal of this out-of-the-box evaluation is not to provide state-of-the-art performance, but to provide an opportunity analysis about the impact of prefix sums on LLMs and demonstrate that more extensive usage of the matrix cores could further improve end-to-end performance in LLMs.

7.1. State-space Neural Networks (SSNN)

The first model architecture that we consider is SSNN, in particular, the Mamba models available on Huggingface with the prefix `state-spaces/mamba-*` (Gu & Dao, 2024). These models are known to rely on efficient parallel scan kernels to achieve good performance. To measure the end-to-end performance opportunity of the Cube-based scan primitive of Mamba on Ascend, we use a simplified, non-optimized version of Mamba known as mamba-tiny (PeaBrane, 2025). We replace the baseline `torch.cumsum` with our optimized kernel (fp32) and compare the inference time for different model sizes. Following (Gu & Dao, 2024), we set the context length to $2,048$ and average the elapsed time over five repetitions. Figure 9 shows a $2.02\times$ speedup, but we expect even stronger benefits with longer context lengths.

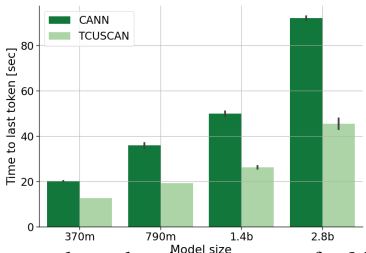

Figure 9. Time to last token improvements for Mamba models with optimized scan in fp32 `ScanU` (TCUSCAN).

7.2. Top-p Token Sampling on LLMs

Although the number of LLMs discussed in the literature is remarkably extensive, their token generation mechanisms includes a handful of sampling strategies such as top-p (nucleus) sampling (Holtzman et al., 2020). Recall that Figure 8 demonstrates a substantial performance improvement when using our optimized top-p operator in isolation. As a next step, it is natural to ask what is the impact of such a performance improvement on end-to-end model inference. We measure the inference time for a set of LLMs where top-p sampling is employed for token generation. As shown in Table 1, top-p has a particularly strong impact on smaller models with large vocabularies, such as BLOOMZ (Muennighoff et al., 2022) and XLM-RoBERTa (Conneau et al., 2019). Our top-p implementation exhibits reduced efficiency for larger models, where the effect of top-p sampling becomes less significant. Additionally, LLMs show to have better performance for larger vocabularies (Takase et al., 2024), and for multi-lingual LLMs having a large vocabulary is a necessity, so we expect larger vocabulary size to be a future trend.

Table 1. Relative inference speedup after optimization across various language models (single batch, prompt length = 1).

Model	Vocab. Size	Model Size	Speedup
Xlm-roberta-base	250,002	278M	$\times$1.47
Xlm-roberta-large	250,002	560M	$\times$1.31
Bloomz-560m	250,680	559M	$\times$1.25
Llama-3.2-1b	128,000	1.24B	$\times$1.05
Qwen3-1.7b	151,643	1.72B	$\times$1.05
Gemma-3-1b-it	262,144	1.00B	$\times$1.04
Mistral-7b-instruct-v0.2	32,000	7.24B	$\times$0.99
GPT2	50,257	124M	$\times$0.97

8. Conclusion

We developed and evaluated efficient parallel scan algorithms tailored to the Ascend architecture by leveraging the power of the cube (matrix multiplication) unit. Our results demonstrate substantial performance improvements, with speedups ranging from $5\times$ to $9.6\times$ compared to vector-only implementations. Additionally, we presented a multi-core Ascend scan algorithm that fully utilizes both the cube and vector units of Ascend, reaching up to 74.9% of the memory bandwidth.

We extended our contributions to include crucial computational kernels for AI workloads, exhibiting significant performance gains. Furthermore, our optimized implementation of radix sort, which utilizes matrix multiplications for parallel splits, showcases the potential of matrix engines , offering up to $3.3\times$ speedup over the baseline. We then provide insight on how these kernels impact some of the most common LLMs, achieving up to $2.02\times$ speedup for Mamba models.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

AMD. AMD CDNA 2 Architecture. Technical report, Advanced Micro Devices, Inc., 2022. URL https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf. Accessed: 2024-09-09.

Baxter, S. moderngpu 2.0. https://github.com/moderngpu/moderngpu/wiki, 2016.

Bell, N. and Hoberock, J. Chapter 26 - thrust: A productivity-oriented library for cuda. In mei W. Hwu, W. (ed.), *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pp. 359–371. Morgan Kaufmann Publishers Inc., Boston, 2012. ISBN 978-0-12-385963-1. doi: 10.1016/B978-0-12-385963-1.00026-5.

Blelloch, G. E. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. doi: 10.1109/12.42122.

Blelloch, G. E. Prefix sums and their applications. In *Sythesis of parallel algorithms*, pp. 35–60. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1990a. ISBN 155860135X.

Blelloch, G. E. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990b. ISBN 026202313X.

Blelloch, G. E. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996. ISSN 0001-0782. doi: 10.1145/227234.227246.

Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)*, SPAA '91, pp. 3–16, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914384. doi: 10.1145/113379.113380.

Blelloch, G. E., Anderson, D., and Dhulipala, L. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)*, SPAA '20, pp. 507–509, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369350. doi: 10.1145/3350755.3400254. URL https://doi.org/10.1145/3350755.3400254.

Chowdhury, R., Silvestri, F., and Vella, F. A computational model for tensor core units. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 519–521, Philadelphia, USA, 2020. ACM. ISBN 9781450369350. doi: 10.1145/3350755.3400252.

Chowdhury, R., Silvestri, F., and Vella, F. Algorithm design for tensor units. In *International Conference on Parallel and Distributed Computing (Euro-Par)*, pp. 353–367, Lisbon, Portugal, 2021. Springer-Verlag. ISBN 978-3-030-85664-9.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., Grave, E., Ott, M., Zettlemoyer, L., and Stoyanov, V. Unsupervised cross-lingual representation learning at scale. *CoRR*, abs/1911.02116, 2019. URL http://arxiv.org/abs/1911.02116.

Dakkak, A., Li, C., Xiong, J., Gelado, I., and Hwu, W.-m. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, ICS '19, pp. 46–57, Phoenix AZ, 2019. ACM. ISBN 9781450360791. doi: 10.1145/3330345.3331057.

DeepSeek-AI. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention, 2025.

Dotsenko, Y., Govindaraju, N. K., Sloan, P.-P., Boyd, C., and Manferdelli, J. Fast scan algorithms on graphics processors. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, ICS '08, pp. 205–213, New York, NY, USA, 2008. ACM. ISBN 9781605581583. doi: 10.1145/1375527.1375559.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Hemstad, J. Lecture 24: Scan at the speed of light. YouTube video, July 2024. URL https://www.youtube.com/watch?v=VLdm3bV4bKo. Online; posted on YouTube.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020. OpenReview.net. URL https://openreview.net/forum?id=rygGQyrFvH.

Hübschle-Schneider, L. and Sanders, P. Parallel weighted random sampling. *ACM Trans. Math. Softw.*, 48(3), September 2022. ISSN 0098-3500. doi: 10.1145/3549934.

Hwu, W. W., Kirk, D. B., and El Hajj, I. Chapter 11 - prefix sum (scan). In *Programming Massively Parallel Processors (Fourth Edition)*, pp. 253–256. Morgan Kaufmann Publishers Inc., Cambridge, Massachusetts, United States, fourth edition, 2023. ISBN 978-0-323-91231-0. doi: 10.1016/B978-0-323-91231-0.00006-9.

Jouppi, N. P. and et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 1–12, Toronto, ON, Canada, 2017. ACM. ISBN 9781450348928. doi: 10.1145/3079856.3080246.

Jouppi, N. P. and et al. Ten lessons from three generations shaped google's TPUv4i: Industrial product. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 1–14, Online – Worldwide, 2021. IEEE. doi: 10.1109/ISCA52012.2021.00010.

Knuth, D. E. *The art of computer programming: sorting and searching*, volume 3. Addison-Wesley Publishing Co., USA, 2nd edition, 1998. ISBN 0201896850.

Kool, W., Van Hoof, H., and Welling, M. Stochastic beams and where to find them: The Gumbel-top-k trick for sampling sequences without replacement. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3499–3508, Long Beach, California, USA, June 2019. PMLR. URL https://proceedings.mlr.press/v97/kool19a.html.

Lakshmivarahan, S. and Dhall, S. K. *Parallel Computing Using the Prefix Problem*. Oxford University Press, Oxford, United Kingdom, 1994. ISBN 9780195358476.

Liang, X. *Ascend AI Processor Architecture and Programming: Principles and Applications of CANN*. Elsevier Science, 2020. ISBN 9780128234891.

Liao, H., Tu, J., Xia, J., and Zhou, X. DaVinci: A scalable architecture for neural network computing. In *Hot Chips: A Symposium on High-Performance Chips (HCS)*, pp. 1–44, Cupertino, CA, USA, 2019. IEEE. doi: 10.1109/HOTCHIPS.2019.8875654.

Liao, H., Tu, J., Xia, J., Liu, H., Zhou, X., Yuan, H., and Hu, Y. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: industry track paper. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 789–801, Seoul, South Korea, 2021. IEEE. doi: 10.1109/HPCA51647.2021.00071.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *MLSys*, pp. 1–14, Santa Clara, California, USA, 2024. mlsys.org.

Merrill, D. and Garland, M. Single-pass parallel prefix scan with decoupled lookback. In *Not available*, pp. 1–9, Santa Clara, CA, USA, 2016. NVIDIA. URL https://research.nvidia.com/.

Meta AI. Llama3 generation code - sample_top_p() method. https://github.com/meta-llama/llama3/blob/main/llama/generation.py#L358, 2024. Accessed: 2024-09-24.

Muennighoff, N., Wang, T., Sutawika, L., Roberts, A., Biderman, S., Scao, T. L., Bari, M. S., Shen, S., Yong, Z.-X., Schoelkopf, H., et al. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*, 2022.

NVIDIA. Cooperative primitives for CUDA C++, 2023. URL https://github.com/NVIDIA/cub.

NVIDIA Authors. NVIDIA DGX-1 with Tesla V100 system architecture. Technical Report MSU-CSE-06-2, Nvidia Corporation, December 2017. URL https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf.

NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*, 2020. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

PeaBrane. mamba-tiny: Simple, minimal implementation of the mamba ssm in one pytorch file. GitHub repository, 2025. URL https://github.com/PeaBrane/mamba-tiny. Available at https://github.com/PeaBrane/mamba-tiny.

Peng, S., Lin, X., Zhang, Y., Xiao, Y., and Hu, Y. Randmscan: accelerating parallel scan via matrix computation and random-jump strategy. *Scientific Reports*, 16:2475, 2025. doi: 10.1038/s41598-025-32283-5. URL https://www.nature.com/articles/s41598-025-32283-5.

Pharr, M. and Fernando, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and*

General-Purpose Computation (GPU Gems). Addison-Wesley Publishing Co., Boston, USA, 2005. ISBN 0321335597.

Rutledge, B. Introducing Coral NPU: A full-stack platform for edge AI, 2025. URL https://developers.googleblog.com/. Accessed: 2025-10-21.

Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for GPU computing. In *Proceedings of the Symposium on Graphics Hardware (EuroGraphics)*, GH '07, pp. 97–106, Goslar, DEU, 2007. Eurographics Association. ISBN 9781595936257.

Smith, J. T., Warrington, A., and Linderman, S. W. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022.

Snir, M. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 7(2):185–201, 1986. ISSN 0196-6774. doi: 10.1016/0196-6774(86)90003-9.

Takase, S., Ri, R., Kiyono, S., and Kato, T. Large vocabulary size improves large language models. *arXiv preprint arXiv:2406.16508*, 2024.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023.

Yan, S., Long, G., and Zhang, Y. StreamScan: fast scan algorithms for gpus without global barrier synchronization. *SIGPLAN Not.*, 48(8):229–238, February 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442539.

Zhang, J., Naruse, A., Li, X., and Wang, Y. Parallel top-k algorithms on GPU: A comprehensive study and new methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '23, New York, NY, USA, 2023. ACM. ISBN 9798400701092. doi: 10.1145/3581784.3607062.

Zhu, H., Cheng, C.-K., and Graham, R. On the construction of zero-deficiency parallel prefix circuits with minimum depth. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2): 387–409, April 2006. ISSN 1084-4309. doi: 10.1145/1142155.1142162.

Zouzias, A. and McColl, W. F. A parallel scan algorithm in the tensor core unit model. In *International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 14100 of *Lecture Notes in Computer Science*, pp. 489–502, Limassol, Cyprus, 2023. Springer. doi: 10.1007/978-3-031-39698-4_33.

A. Appendix: Reproducibility

A.1. Code and Data Availability

All source code, datasets, and scripts required to reproduce the functional correctness of all AscendC kernels presented in this paper are available at:

- **Code repository:** https://gitlab.com/faceless-research/ascendc-scan
- **Dataset:** All datasets are synthetic and randomly generated using *numpy*, *scipy*, or *PyTorch*.

A.2. Environment and Dependencies

- **Operating System:** Ubuntu 20.04
- **GCC version:** 11.4.0
- **Python version:** 3.10
- **Dependencies:**
  - NumPy 1.26.4 / SciPy 1.13.1
  - Ascend CANN toolkit 8.2.rc1.alpha002-910b
- **Environment management (docker):** The publicly available Docker image quay.io/ascend/cann:8.2.rc1.alpha002-910b-ubuntu22.04-py3.10 can be used to verify the correctness of our algorithms using CPU simulation.

To set up a docker container with all required dependencies, do

```
docker run -ti quay.io/ascend/cann:\
  8.2.rc1.alpha002-910b-ubuntu22.04-py3.10 \
  /bin/bash
apt update
apt install -qy cmake python-is-python3 git
git clone \
  https://gitlab.com/faceless-research/\
  ascendc-scan
cd ascendc-scan
```

A.3. Functional Correctness

To build and execute an AscendC kernel, the main entry-point is a single Makefile. To verify the functional correctness of an AscendC kernel, say `scan_single_core`:

- **Functional correctness check (CPU/NPU):**

```
make run_scan_single_core_cpu
make run_scan_single_core_npu
```

The relevant kernels are listed at `src/kernels/kernel_*.h`. For NPU functional correctness, an Ascend 910B is required. Tested on 910B4. Set DEVICE=Ascend910B2 variable on Makefile for 910B2. For detailed instructions, see the README.md file in the root folder of the repository.

A.4. Computational Resources

- For NPU, experiments were run on a single Ascend 910B4 AI Accelerator having 24GB HBM memory.
- Average time per kernel run: 1 minute.
- Number of iterations per experiment: 100

A.5 Continuous Testing and Documentation

We use GitLab CI infrastructure for continuous testing, see https://gitlab.com/faceless-research/ascendc-scan/-/pipelines.

We used doxygen for code documentation, type `make docs`. A documentation instance is available at https://ascendc-scan-247217.gitlab.io/files.html.

A.6 Additional Notes

We fix all random seeds (‘numpy‘, ‘scipy‘, and ‘random‘) to ensure deterministic behaviour.

B. AscendC Programming Model

Recently, a pipeline-based programming model for Ascend called *AscendC* has been developed. AscendC allows its users to build high-performance computational kernels for the Ascend architecture. Such kernels are usually called AscendC operators. The AscendC programming model is built on top of C++, and it allows its users to have fine-grained control of Ascend's hardware components, such as MTEs, scalar, vector, and cube compute engines. At the same time, AscendC eliminates many potential problems, such as the need to explicitly synchronize hardware components within AIC and AIV cores.

The AscendC programming model is based on a multiple pipeline abstraction model. AscendC provides users with some abstractions, including a context manager object, tensors, queues, and buffers.

AscendC provides tensor structures as wrappers over data allocated in the global or core's local memory. *GlobalTensor* is a structure that represents a buffer in global memory. In all operators, both input and output data come from global tensors. On the other hand, *LocalTensor* represents a buffer in the core's local memory. Users can allocate local tensors in one of multiple possible hardware buffers (UB, L1, L0A, L0B, etc.).

AscendC also provides the *queues* API – data structures used for managing tensors and resolving data dependencies between different hardware components which work on the same tensors. After a hardware component interacts with a tensor the `Enque` method is called, that saves the pointer into the queue. Then when the next hardware component needs to interact with the same tensor the `Deque` method is called, that waits for the corresponding `Enque` to be called, ensuring synchronization, and returns the pointer to the local tensor. This way, all data dependencies are explicit, and the computational pattern is consistent across all local tensors and all physical buffers. Queues can contain more than one tensor at a time – in many cases, implementing double buffering comes down to changing the queue capacity from the default value one to two.

Naturally, the model also defines dozens of possible operations on tensors; we only mention some of the most frequently used here:

- *DataCopy*. MTE's function that copies data from an input to an output tensor. The basic version copies a number of continuous elements but can also be configured for strides and automatic layout transformations.
- *Mmad*. AIC core's function that multiplies two input matrices (local tensors) and writes the result to the output matrix (a local tensor). The result can be accumulated with existing values in the output tensor.
- *Adds*. AIV core's function that adds a scalar to an input local tensor and writes the result to the output local tensor.
- *GatherMask*. AIV core's function that takes an input local tensor and a binary mask, also a local tensor, and gathers all the elements from the input tensor for which the corresponding value in the mask is equal to 1. Gathered elements are stored in the contiguous form in the output local tensor.

An operator is executed using multiple *blocks* – block is the smallest logical execution unit. The user specifies the number of blocks to be used when running the kernel. Another critical function AscendC provides is hardware synchronization among computing units – *SyncAll* allows the user to synchronize all blocks. The execution is continued only after each unit reaches the synchronization point.

C. Tensor Core Unit (TCU) Model

The Tensor Core Unit model is a standard RAM model with an additional circuit, named tensor core unit, that per-

forms matrix multiplication between constant-size matrices (Chowdhury et al., 2020). Although the TCU model captures well a single matrix multiplication computational unit of today's accelerators, it ignores other essential features: the presence of vector processing units and, more critically, the multi-core nature of these accelerators. Since the TCU model considers only a single matrix multiplication unit, it does not allow its users to conduct a work/depth algorithmic analysis (Blelloch, 1996). Due to the above limitations, any algorithmic analysis in the TCU model will not correspond to a realistic execution in Ascend, i.e., ignoring parallelism and the vector units. Nevertheless, we discuss the work/depth asymptotic analysis of the proposed algorithms, assuming the presence of multiple matrix engines and vector units, considering their operations as basic operations.

Work/Span Analysis of Single Core Scans. We bound the work and span of the single-core algorithms. For work, we separately report the number of square matrix multiplications of size s and the sum of lengths of the vector instructions similar to the Vector-RAM model (Blelloch, 1990b). The number of matrix multiplications/accumulations for `ScanU` and `ScanUL1` are $\lceil n/s^2 \rceil$ and $3\lceil n/s^2 \rceil$, respectively. The total vector work for both algorithms is $\mathcal{O}(n)$.

Next, we bound the span of the algorithms. The number of vector instructions called by the algorithms `ScanU` and `ScanUL1` are $\mathcal{O}(n/s)$ and $\mathcal{O}(n/s^2)$, respectively. Notice that the vector length of the `ScanUL1` vector operations is $\mathcal{O}(s^2)$. Since the vector instructions are on the critical path, the span of `ScanU` and `ScanUL1` is $\mathcal{O}(n/s)$ and $\mathcal{O}(n/s^2)$, respectively. In summary, `ScanUL1` has a $\mathcal{O}(s)$-depth speed-up compared to `ScanU`, but, in practice, the achievable speed-up is ≈ 1.92 for $s = 128$, see Figure 4.

Work/Span Analysis of `MCScan`. First, we bound the work of `MCScan` in terms of the number of matrix multiplications of size s, and the number of vector instructions (scalar-vector addition) of length s. The number of matrix multiplications of `MCScan` is $\lceil n/s^2 \rceil$. The vector instructions of length s consist of $\mathcal{O}(n/s)$ reductions in the first phase, and $\mathcal{O}(n/s)$ scalar-vector additions in the second phase. Hence, the total number of vector operations is $\mathcal{O}(n/s)$. The critical path of `MCScan` is on the second phase where the vector units propagate the partial sums, thus the span of `MCScan` is $\mathcal{O}(\frac{n}{sB})$.

D. Additional Material for Scans

D.1. Multiple (Batched) Scans

The batched scan computes the prefix sum of a batch of input arrays of equal length in parallel. Given a scan algorithm for a 1D array, a corresponding batched scan algorithm could be defined by deciding how to schedule the cube and vector computations into the multiple cores.

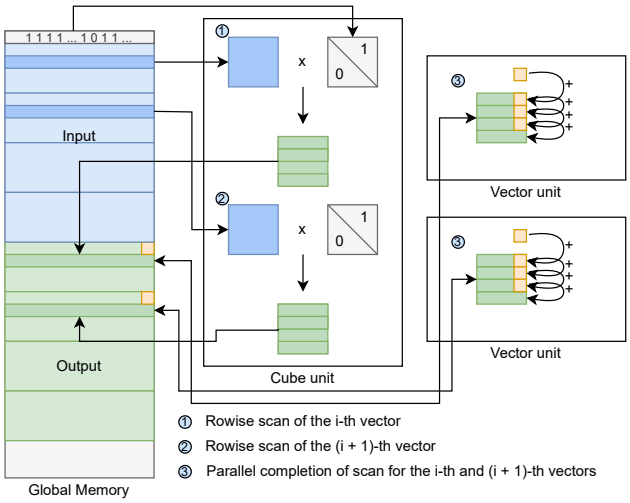

Figure 10. Batched scan algorithm based on `ScanU`. Blue and green denote the input and output array, respectively.

Let us discuss, as an example, a particular case where `ScanU` is used as a building block. The batched algorithm uses the same principles as the Algorithm 4.1 but also considers the 2-to-1 ratio between the vector and cube cores in the split Ascend architecture (910B). Figure 10 depicts the main ideas behind our batched scan algorithm in this case. The algorithm starts by computing the local scans of size s of $\boldsymbol{x}$ of all input arrays. Each cube core computes the local scans of a tile of size s^2 in two batches at the same time; once the tiles of the first two batches are ready, two distinct vector cores will complete the scans independently over each batch by propagating the partial sums within the tiles. This process is pipelined through AscendC to efficiently use all available hardware (vector) resources. The second batched scan algorithm extends `ScanUL1` (Algorithm 4.2) so that each AI core computes a scan on a separate array in the batch.

Figure 11 compares the two batched scan algorithms presented for increasing input batch size and array length. The first batched scan algorithm, based on `ScanU` (Algorithm 4.1), is used as our reference/baseline. Both algorithms have the same tiling strategy based on the input shapes to ensure a fair comparison. The figure demonstrates that the algorithms perform well in different cases and, more importantly, complement each other. In particular, `ScanU` is superior when the batch size is greater than 18, and the input length is smaller than $4K$. On the other hand, `ScanUL1` is superior when the batch size is smaller than 18 and the input length larger than $4K$.

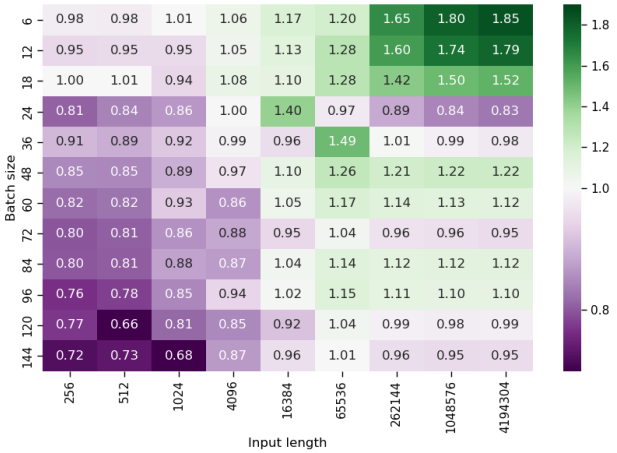

Figure 11. Execution time ratio between `ScanUL1` and `ScanU` batched scan algorithms for various array length (x-axis) and batch sizes (y-axis). Baseline is `ScanU`.

D.2. Exclusive and int8 scan

Here, we discuss a few extensions of the multi-core scan algorithm that we have implemented. In particular, we added support for exclusive scans and integer inputs. Typically, AI accelerators support low-precision arithmetic since inference of deep learning models is robust to extreme levels of quantization (Lin et al., 2024). In particular, Ascend supports input matrices of 8-bit integers with output/accumulation in 32-bit integers. Since scan is a memory-bound operator, there is an opportunity to improve performance in terms of elements per second processed; see Figure 14. We have implemented a specialization of Algorithm 4.3 for integers with 8 bits, which is extensively used in our scan applications.

We implemented exclusive scan by writing the output of the inclusive scan to global memory shifted by one element, discarding the last value and writing zero to the first position using the first block.

D.3. Batched Scan

Figure 12 depicts the performance of the batched scan kernels on Ascend for varying input batch sizes and input lengths equal to $65K$. We depict the memory bandwidth achieved for tiling parameters $s = 16, 32, 64$ and 128. Our proposed batch scan operators for $s = 64$ and 128 reach up to 400 GB/s. Interestingly, for smaller values of $s = 16, 32$, the performance of the proposed batch scan kernels is poor. In addition, the performance of our proposed batch scan kernel for $s = 16$ and the baseline is similar.

D.4. Compress

Figure 13 depicts the performance comparison between compress versus the baseline PyTorch `masked_select`.

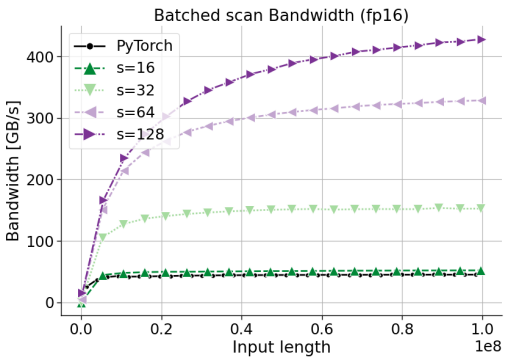

Figure 12. Bandwidth of batched scan based on Algorithm 4.1 for increasing batch sizes and $s = 16, 32, 64, 128$. Input length is $65K$.

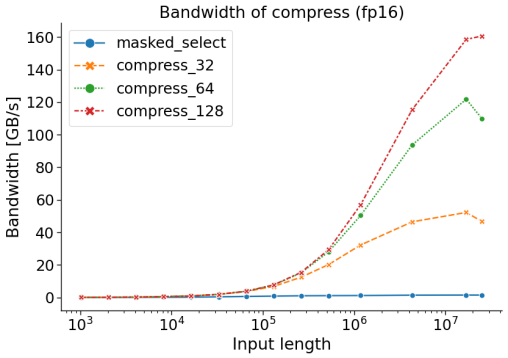

Figure 13. Bandwidth of compress operator based on `MCScan` ($s = 32, 64, 128$) against `torch.masked_select`.

We set the mask so that each mask entry is independently set to true or false uniformly at random. The figure indicates that the baseline `masked_select` operator is not optimized on Ascend, and a code investigation reveals that the baseline does not use the vector or cube units. On the other hand, our Compress kernel reaches up to 160GB/s (20% of peak memory bandwidth).

D.5. Weighted Sampling

We implement a parallel weighted sampling kernel using the well-known inverse transform sampling approach and a scan to compute the cumulative distribution. Given an array $\boldsymbol{w}$ of n positive weights, the goal is to draw a sample with proportional probability to the weights. The output is an index i of $\boldsymbol{w}$ with probability proportional to w_i. First, we scan $\boldsymbol{w}$, and then, given a uniform sample $\theta \in [0, 1]$, we invoke the `SplitInd` kernel with input scan($\boldsymbol{w}$) and the element-wise predicate $? > \theta * \sum_i w_i$. The last entry of the output indices array of `SplitInd` contains the weighted sample. For more advanced parallel weighted sampling

techniques, see (Hübschle-Schneider & Sanders, 2022).

The performance improvement of our proposed weighted sampling kernel is not significant compared to the baseline for a single sample. However, our implementation does provide a functional improvement compared to the baseline operator. Indeed, the baseline Ascend weighted sampling operator `torch.multinomial` supports discrete distributions with support size up to 2^{24} elements, whereas our approach can support distributions with arbitrary support size. We leave as future work any further possible improvements on parallel weighted sampling. In particular, for the multiple sample generation scenario, the parallel alias table construction of (Hübschle-Schneider & Sanders, 2022) seems to be a promising direction.

D.6. Scan on low bit-width data types

Next, we investigate the additional performance benefits of taking advantage of the lower-precision input data (int8) capability of the cube unit. Figure 14 depicts the performance of the multi-core scan algorithm in terms of giga elements per second for input data types float16 and 8-bit integers (int8). As depicted, there is only a small performance improvement for relatively small length, which is mainly due to our current implementation. For example, additional performance can be achieved since the Cube unit can multiply, in a single instruction, matrices of sizes 128×256 and 256×128 in int8.

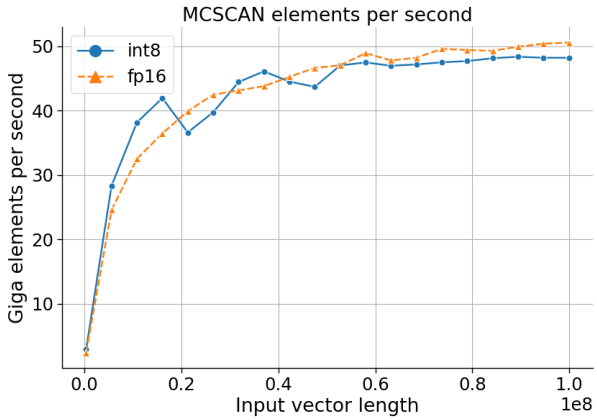

Figure 14. Giga elements per second comparison of `MCScan` for float16 (fp16) and 8-bit integers (int8) input data types.

D.7. Radix sort on low bit-width inputs

Here, we provide an opportunity estimate of the expected performance improvements of radix sort for the case where the input data type has bit-width 4 or 8. To do so, we measure the execution time of our radix sort kernel by only reducing the number of radix sort iterations from 16 to 4 or 8 (algorithmic correctness is not guaranteed). Notice that we still read and write 16 bits per element. Figure 15 shows a significant performance improvement of roughly $1.5\times$ and $3\times$ of our radix sort kernel for 8 and 4 bits inputs, respectively. The performance peak around 10M elements is due to the L2 cache. An interesting future research direction is to make our radix sort cache-aware.

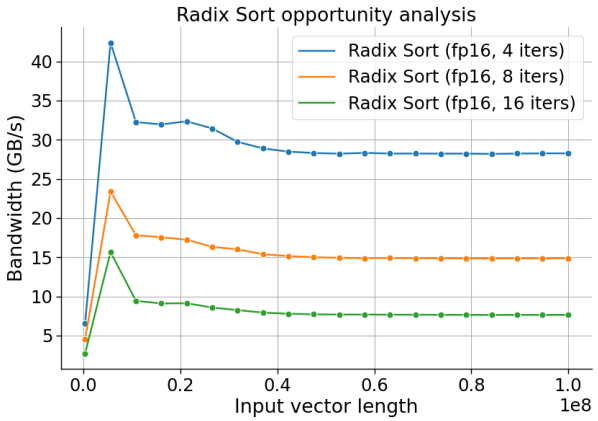

Figure 15. Radix sort opportunity analysis for low bit-width inputs.