Parallel Scan on Ascend AI Accelerators

BARTŁOMIEJ WRÓBLEWSKI*, GIOELE GOTTARDO*, and ANASTASIOS ZOUZIAS, Computing Systems Lab, Huawei Zurich Research Center, Switzerland

We design and implement parallel prefix sum (scan) algorithms using Ascend AI accelerators. Ascend accelerators feature specialized computing units—the cube units for efficient matrix multiplication and the vector units for optimized vector operations. A key feature of the proposed scan algorithms is their extensive use of matrix multiplications and accumulations enabled by the cube unit. To showcase the effectiveness of these algorithms, we also implement and evaluate several scan-based operators commonly used in AI workloads, including sorting, tensor masking, and top-k / top-p sampling.

Our single-core results demonstrate substantial performance improvements, with speedups ranging from 5× to 9.6× compared to vector-only implementations for sufficiently large input lengths. Additionally, we present a multi-core scan algorithm that fully utilizes both the cube and vector units of Ascend, reaching up to 37.5% of the theoretical memory bandwidth. Furthermore, our radix sort implementation, which utilizes matrix multiplications for its parallel splits, showcases the potential of matrix engines to enhance complex operations, offering up to 3.3× speedup over the baseline.

CCS Concepts: • **Theory of computation → Parallel algorithms**.

Additional Key Words and Phrases: prefix sum, scan, matrix multiplication, matrix engines, tensor cores, Ascend, AI accelerators

ACM Reference Format:
Bartłomiej Wróblewski, Gioele Gottardo, and Anastasios Zouzias. 2025. Parallel Scan on Ascend AI Accelerators. In *Proceedings of Unpublished manuscript (Conference acronym 20XX).* ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

arXiv:2505.15112v1 [cs.DC] 21 May 2025

1 Introduction

Parallel scan is a fundamental parallel computing paradigm with many applications [7, 28]. 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) model [8]. In the circuit model, the depth and size trade-offs for parallel optimal prefix circuits are well-understood for binary operations [44, 50], but other models are yet to be explored, especially in the heterogeneous computing domain [12, 51].

Despite many attempts, there is a large gap between abstract parallel machine models and the current state-of-the-art accelerators that contain heterogeneous computing units, see[35] and [2, 25, 43]. For example, hardware vendors have manufactured accelerators with specialized compute engines known as *matrix engines* or *tensor core units*. A list of such specialized hardware units includes Google's TPUs [22, 23], Nvidia's Tensor Cores [39], AMD's Matrix Cores [1] and Huawei's Ascend Cube Unit [32, 33] to name a few. Moreover, in the CPU domain, examples of such units (or extensions) are ARM's Scalable Matrix Extension (SME) [3], Intel's Advanced Matrix Extensions (AMX) [20] and IBM's POWER10 Matrix-Multiply Assist (MMA) [46]. Therefore, today's high-performance processors contain matrix engines that allow efficient multiplication of

*Work done while the authors were employed at Huawei Zurich Research Center.

Authors' Contact Information: Bartłomiej Wróblewski, bartek.wroblewski@huawei.com; Gioele Gottardo, gioele.gottardo@huawei.com; Anastasios Zouzias, anastasios.zouzias@huawei.com, Computing Systems Lab, Huawei Zurich Research Center, Zurich, Switzerland.

Conference acronym 20XX, May 2025, N/A
2025. ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

constant-sized matrices, making ideas from the early 1980s [27, 45] a reality and initiating fruitful debates within the community [13].

Given the presence of matrix engines in computing products, a model of computation was recently proposed to capture matrix multiplication accelerators called the Tensor Core Unit (TCU) model [10, 11]. The authors of [11] initiated the study of TCU algorithms and revisited classical paradigms in the TCU model. From the perspective of hardware-aware algorithmic implementations, the seminal paper of [12] proposed and implemented parallel scan (and reduction) kernels tailored to GPU accelerators. We drew inspiration from both these works [11, 12] to advance the study of parallel scans on matrix multiplication accelerators. 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 [31].

Our main contributions are listed below:

- We design, implement, and evaluate parallel scan algorithms specialized for the Ascend AI accelerator. The distinguishing feature of the proposed algorithms is the extensive use of the Ascend matrix multiplication engines ("Cube cores"). 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 unit's 8-bit integer capabilities.
- We evaluated the proposed scan algorithms that use a single cube and vector units, demonstrating a 5× up to 9.6× speed-up compared to the baseline scan (vector unit only) for sufficiently large input lengths.
- We present a multi-core scan (`MCScan`) algorithm that fully utilizes both the cube and vector units of the Ascend 910B4 accelerator, reaching up to 37.5% of the theoretical memory bandwidth. `MCScan` achieves a 15.2× speedup compared to the single cube algorithm when it uses all available (20) cube cores and vector cores.
- We implement a list of computational kernels (operators), including parallel split, compress/-compact and top-p (nucleus) sampling essential to AI workloads. In all of these cases, we demonstrate significant performance improvements.
- Lastly, we implement radix sort, whose parallel splits take advantage of the cube units. The radix sort implementation provides up to 3.3× speed-up over baseline sorting that does not utilize the cube units.

Next, we briefly emphasize the key distinction between the above contributions and prior art. First, the multi-core scan algorithm is novel (to the best of our knowledge) since it performs partial recomputation of the reduction values on both the cube and vector units in its first phase. This recomputation strategy is different compared to all previously known scan strategies on accelerators, see Section 2.1. Such a recomputation strategy could be of interest to other matrix multiplication accelerators as well.

Second, our radix sort implementation yields an intriguing result: in practice, multiple small dense matrix multiplications can be leveraged to improve the end-to-end performance of parallel sorting. Although the algorithmic ideas and techniques underlying this result are well-established [9], we believe that it paves the way for interesting research directions in the future. As an example, we pose the following question. Is it possible to utilize the multiple-add capabilities of the matrix multiplications units to improve further the performance of parallel sorting? A similar question of "Can we sort with matrix multiplications?" was raised during the presentation of [10]. Note that although the parallel splits of our radix sort implementation heavily use matrix multiplications to perform scans, they do not use the multiply-add capabilities of the matrix multiplication unit.

2 Background

In this section, we give an overview of related work in computing parallel scans using accelerators, focusing mostly on matrix multiplication accelerators. We also present the Tensor Core Unit (TCU) model of computation, and we give a summary of a selected list of scan-based primitives. The (inclusive) prefix sum or scan of a sequence is a new sequence where each term at index i is the cumulative sum of all terms from the beginning of the original sequence up to index i.

2.1 Scan Strategies on Accelerators

Numerous parallel scan implementations and libraries have been proposed in the literature [4, 5, 14, 18, 36, 38, 42, 48]. Here, we discuss the most relevant work focusing on scan implementations targeting accelerators, most notably Graphics Processing Units (GPUs). Horn was one of the first to implement parallel scans in GPUs [41, Chapter 36], 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 [36]. Scan-Scan-Add (SSA) means that the (local) *scans* are initially computed per block. Second, the values of the largest index of each block are collected and *scanned*. Third, the collected per-block scan values are broadcast-*added* to their corresponding blocks. Similarly, the RSS approach makes a block-level reduction first, followed by a scan of the block-level reductions and a final scan of each block. We refer the interested reader to [36, Section 3] for a more detailed discussion.

Scan is a memory-bound operation and, hence, the main drawback of the SSA and RSS scan strategies is the high number of elements that are read and written to global memory. In particular, for input length N, SSA reads/writes $\approx 4N$ elements, whereas RSS reads/writes $\approx 3N$ elements. Such a reduction in the memory access size is critical for improving the performance of scans.

StreamScan and decouple look-back strategies access only $2N$ memory elements but need to efficiently handle the sequential data dependency of the scan computation using adjacent (block) synchronization, see [19, Chapter 11.7]. StreamScan is a single-pass approach in which each thread block is assigned a tile of input, and a serial dependency between the blocks exists [48]. The critical feature of StreamScan is that it requires synchronization between adjacent blocks only (without global block-level synchronization). The decoupled look-back strategy of [36] aims to alleviate the drawbacks of the serial dependency of StreamScan by performing redundant work to "dissociate" local computation from the latency of global prefix propagation. Both StreamScan and decouple look-back strategies require only $2N$ data movement to global memory: N input elements are read, N output elements are written.

2.2 Scan using Matrix Multiplications

The study of accelerating prefix sum (and reduction) operations using matrix multiplication units was first initiated in the seminal paper of [12]. The authors of [12] designed scan algorithms for the GPU architecture by providing highly optimized CUDA kernels and, hence, use the terms of warp/block/grid of the CUDA programming model. They proposed a warp-level scan algorithm [12, Algorithm 6], and a block-level scan algorithm [12, Algorithm 7]. Moreover, they mentioned that the device/grid level algorithm is based on a textbook approach [18].

A follow-up work presented a parallel scan algorithm in the TCU model of computation where only matrix multiplication operations are required and, more importantly, the depth of the computation is logarithmic in the input size [51]. Although the main algorithm of [51] has linear work

and logarithmic depth/span, its strided memory access patterns are typically hard to translate to efficient memory access operations.

2.3 Tensor Core Unit (TCU) Model

To the best of our knowledge, the Tensor Core Unit (TCU) model is the only model of computation that has been recently proposed to capture matrix multiplication accelerators [10]. The TCU model is a standard RAM model with an additional circuit, named tensor core unit, that performs matrix multiplication between constant-size matrices. 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 [8]. 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.

2.4 Applications of Parallel Scan

Parallel scan has a plethora of applications [7, 28]. 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, weighted sampling, 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).

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 [33]. Moreover, we discuss the AscendC programming model of Ascend, a recently proposed programming model for Ascend operator development. All the material presented here is available online at https://www.hiascend.com.

3.1 Ascend Hardware

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 and more complex ones such as gather and reduce.

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

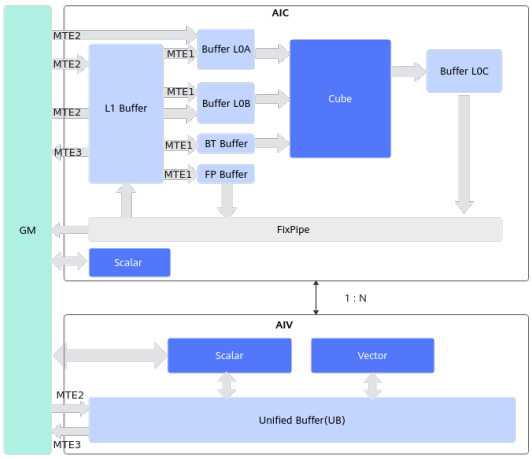

Fig. 1. Architecture of Ascend 910B Training series accelerators. Each AI core contains one cube and two vector units.

computing engine. An AIC core can be configured to multiply two matrices of almost arbitrary sizes. It also supports result accumulation, selected activation functions, and quantization operations. The cube core supports both floating point and low-precision integer data types, i.e. float16 (with float32 output) and int8 (with int32 output).

Figure 1 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. This approach dramatically simplifies the implementation logic, but each data transfer between the AIC and AIV cores might be expensive in terms of performance.

3.2 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.

4 Scan Algorithms on Ascend

In this section, 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.

Matrix multiplication is an essential operator in our discussions 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., $\boldsymbol{A}_s$, and define $\ell := s^2$. 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. $\boldsymbol{L}_s$ 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 ℓ, i.e., tiles are contiguous blocks of ℓ entries of $\boldsymbol{x}$. We note an arbitrary ℓ-tile of $\boldsymbol{x}$ by $\boldsymbol{x}_\ell$. We also view a tile $\boldsymbol{x}_\ell$ as a row-major matrix $\boldsymbol{A}$ having s rows and s columns (pad with zeroes if necessary).

In Section 4.1, we present two scan algorithms (Algorithm 1 and Algorithm 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 2 performs multiple matrix multiplications and utilizes the accumulation buffer of the Cube unit to compute the scan of an input tile of length ℓ, whereas Algorithm 1 computes s consecutive scans of smaller tiles of length s using a single matrix multiplication.

Next, in Section 4.2, we build on top of the ScanU and ScanUL1 algorithms and extend them into batched scan variants that operate on multi-dimensional arrays (tensors). Here, we discuss issues that arise when scheduling several scan operations into multiple cores, including padding, better load balancing on vector/cube units with a ratio of 2:1, etc.

Last but not least, in Section 4.3, we present a multi-core scan algorithm (MCScan, Algorithm 3) tailored for the Ascend AI accelerator. A key feature of MCScan is that it utilizes all the available cube and vector cores. MCScan is designed for scenarios involving very large one-dimensional arrays.

4.1 Warm-up: single cube scans

In this section, 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 $\boldsymbol{A}$ @ $\boldsymbol{U}_s$ computes "local" scans of tiles of size s of $\boldsymbol{x}$.

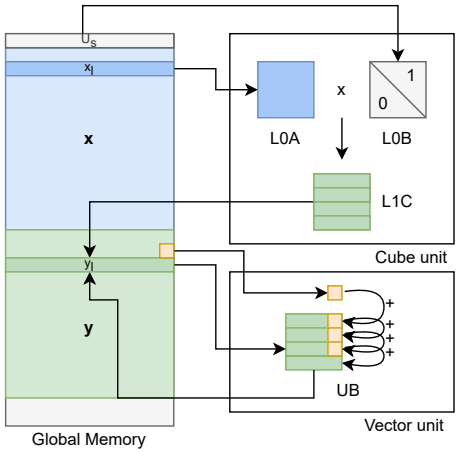

Fig. 2. Diagram that shows the data path from an input tile $\boldsymbol{x}_\ell$ to an output tile $\boldsymbol{y}_\ell$ of the ScanU (Algorithm 1). Blue and green denote the input and output array, respectively.

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 developing 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 4.2 and 4.3.

The first algorithm, ScanU (Algorithm 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. More precisely, once the cube unit has computed the local row scans of a matrix tile of size ℓ, the tile is sent to a vector core for further processing. The vector core will add a scalar to each row to correct the prefix sum. It is important to note that to obtain the correct prefix sum, the vector core keeps track of the last value of each ℓ-tile and propagates it to the next tile.

Figure 2 depicts the data movements and the memory view of ScanU. The input vector $\boldsymbol{x}$ is shown in blue. A tile $\boldsymbol{x}_\ell$ of $\boldsymbol{x}$ and $\boldsymbol{U}_s$ are loaded into the cube unit where the matrix multiplication occurs. The matrix multiplication result is written to global memory. The vector unit reads the

cube unit output tile and propagates the prefix sum in place. The whole process, which consists of memory transfers and cube/vector operations, is pipelined over the input tiles using AscendC software pipeline capability.

Algorithm 1 Scan Cube-Vector (ScanU)

```
procedure ScanU(x, s)
    Let y be the output array
    partial ← 0                                   ▷ Accumulation value (Vector Unit)
    Load U_s in L0B
    for each s^2-tile of x: x_ℓ do                ▷ Pipelined execution
        Load x_ℓ to L0A
        C ← A_s @ U_s                             ▷ acc. off, free L0A
        Copy C from L0C to y_ℓ in GM
        Vector core waits for Cube core
        Copy y_ℓ from GM to UB                    ▷ Vector unit
        for each s-tile of y_ℓ: y_s do
            y_s ← y_s + partial                   ▷ In-place add
            partial ← last entry of y_s
        end for
        Copy y_ℓ from UB to GM
    end for
    Return y
end procedure
```

The second algorithm, ScanUL1 (Algorithm 2), is an Ascend adaptation of [12, Algorithm 6] and is based on a matrix identity that expresses the scan of an array z of length ℓ using matrix operations. View z as a square row-major matrix A of size $s = \lceil \sqrt{\ell} \rceil$ (pad with zeroes if needed). Given z, the inclusive scan of z ($\mathsf{scan}(z)$) can be computed as:

$$\mathsf{scan}(z) = A_s \mathbin{@} U_s + L_s^- \mathbin{@} A_s \mathbin{@} \mathbf{1}_s, \tag{1}$$

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

$$\begin{aligned} C_1 &= A_s \mathbin{@} \mathbf{1}_s \\ C_2 &= A_s \mathbin{@} U_s \\ C_2 &= C_2 + L_s^- \mathbin{@} C_1. \end{aligned}$$

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 A, allowing us to load A only once in L0A. Moreover, the third step effectively utilizes the accumulation buffer of the cube unit since C_2 is reused in the last two steps. Once the local scan of a tile of size ℓ 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 2).

Algorithm 2 ScanUL1 is an Ascend adaptation of [12, Algorithm 6]

```
1:  procedure ScanUL1(x, s)
2:      Let y be the output array
3:      partial ← 0                                   ▷ Accumulation value (Vector Unit)
4:      Load U_s, L_s^-, 1_s in L1
5:      for each s^2-tile of x: x_ℓ do                ▷ Pipelined execution
6:          Load x_ℓ to L0A and 1_s to L0B
7:          C_1 ← A_s @ 1_s                           ▷ acc. off, no free inputs
8:          Copy C_1 from L0C to L1
9:          Load U_s to L0B
10:         C_2 ← A_s @ U_s                           ▷ acc. off, no free inputs
11:         Load L_s^- in L0A and C_1 in L0B
12:         C_2 ← C_2 + L_s^- @ C_1                   ▷ acc. on, free all buffers
13:         Copy C_2 from L0C to y_ℓ in GM
14:         Vector unit waits for cube unit
15:         Copy y_ℓ from GM to UB                    ▷ Vector unit
16:         y_ℓ ← y_ℓ + partial
17:         partial ← last entry of y_ℓ
18:         Copy y_ℓ from UB to GM
19:     end for
20:     Return y
21: end procedure
```

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] with `CumSumInfo` parameters set to 128 and 128. We also set $s = 128$ on the cube scan algorithms to ensure a fair comparison. Figure 3 compares the cube scan algorithms and the vector-only algorithm provided by AscendC. The figure demonstrates a significant performance improvement (5× for `ScanU`, and 9.6× for `ScanUL1`) compared to the vector-only `CumSum` algorithm. Moreover, the figure shows that the `ScanUL1` scan algorithm has roughly a 2× speedup compared to `ScanU`. The critical insight here is that a more sophisticated usage of the computational capabilities of the cube unit can deliver further significant performance improvements.

4.2 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.

Let's 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 1 but also considers the 2-to-1 ratio between the vector and cube cores in the split Ascend architecture (910B). Figure 4 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 ℓ in two batches at the same time; once the tiles of the first two batches are ready, two distinct

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

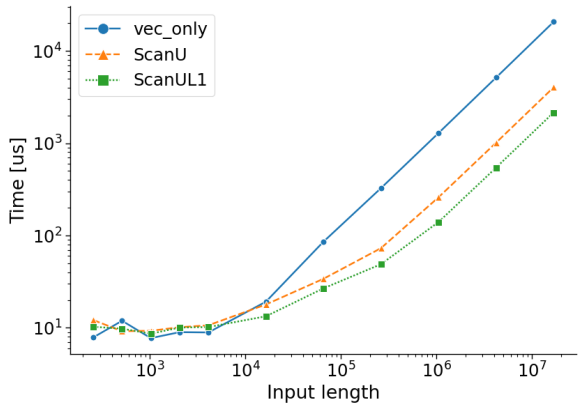

Fig. 3. Execution time of CumSum AscendC API (vec_only) versus ScanU and ScanUL1 (log-log scale).

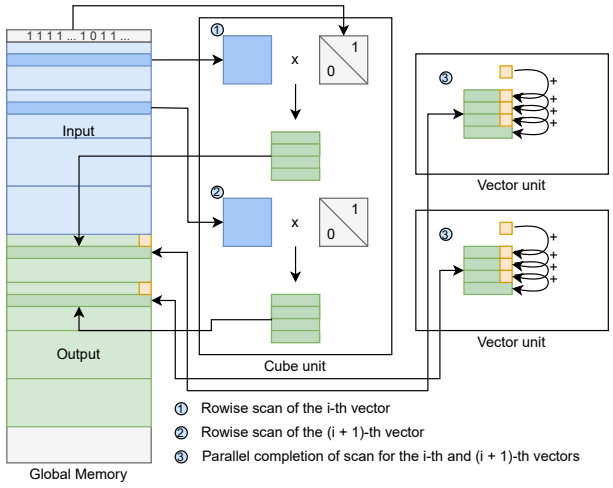

Fig. 4. Batched scan algorithm based on ScanU. Blue and green denote the input and output array, respectively.

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 2) so that each AI core computes a scan on a separate array in the batch. Figure 5 compares the two batched scan algorithms presented for varying input batch size and array length. The first batched scan algorithm, based on ScanU (Algorithm 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 $4K$.

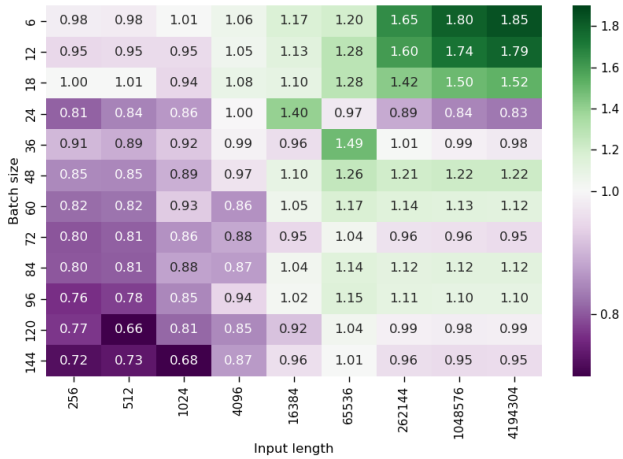

Fig. 5. Execution time ratio between ScanUL1 and ScanU batched scan algorithms for various array length (x-axis) and batch sizes (y-axis). Baseline is ScanU.

4.3 Multi-core Scan

This section presents a multi-core scan algorithm (MCScan, Algorithm 3) to compute the prefix sum of large input arrays. For the sake of a clearer explanation, the presented algorithm uses a vector-to-cube ratio of 1-to-1. However, 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). Figure 6 depicts these two phases of the algorithm. In contrast to previous methods, MCScan performs partial re-computation of the block-level reductions during its first phase, as we will explain shortly.

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 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 sum 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.

Exclusive scan and int8 support. 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 [34]. In particular, Ascend supports input matrices of 8-bit integer data type with output/accumulation in 32-bit integer data type. Since the scan is a memory-bound operator, there is an opportunity to improve performance in terms of elements per

Algorithm 3 Multi-core Scan (`MCScan`)

```
1:  procedure MCScan(x, s, B)                                  ▷ B: number of blocks
2:      Let y be the output array
3:      Let r be an array of length B in GM
4:      parfor i-th block of x: x[i] do                        ▷ Phase I
5:          Load U_s in L0B                                    ▷ Cube Units
6:          for each ℓ-tile of x[i]: x_ℓ do
7:              Load x_ℓ from GM to L0A                        ▷ Cube Units
8:              C ← A_s @ U_s                                  ▷ Cube Units
9:              Copy C in y[i] in GM                           ▷ Cube Units
10:         end for
11:         Load x[i] to UB                                    ▷ Vector Units
12:         r_i ← ReduceSum(x[i])                              ▷ Vector Units
13:         Write r_i on i-th entry of r in GM                 ▷ Vector Units
14:     end parfor
15:     SyncAll: Synchronize all cube/vector cores
16:     parfor i-th block of y: y[i] do                        ▷ Phase II
17:         Load r from GM to UB
18:         partial ← Sum first i entries of r
19:         for each ℓ-tile of y[i]: y_ℓ do
20:             for each s-tile of y_ℓ: y_s do
21:                 y_s ← y_s + partial                        ▷ In-place add
22:                 partial ← last entry of y_s
23:             end for
24:             Copy y_ℓ from UB to GM
25:         end for
26:     end parfor
27:     Return y
28: end procedure
```

second processed; see Figure 9. We have implemented a specialization of Algorithm 3 for integers with 8 bits.

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

5 Operators based on Scan

In this section, we revisit several computational parallel primitives based on parallel scans [6, 42]. Scan-based primitive include weighted sampling, split and compress/compact (equivalent to the `masked_select` Pytorch operator). It is well-known that radix sort can be implemented on top of split [7, Section 1.3]. Also top-k can be implemented on top of split using a partial quick-sort/select approach [8].

Interestingly enough, 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 [16], see also [15]. The top-p sampling implementation of the popular open-source model Llama3 [47] contains a batched sorting and prefix sum operation as the first two PyTorch operations, see [37].

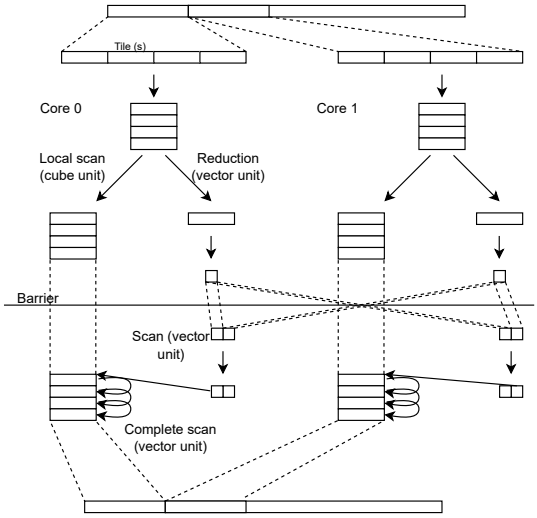

Fig. 6. Multi-core scan (Algorithm 3) consists of two phases synchronized by a global barrier. The first phase computes tile-level (size s) local scans on cube units and block-level reductions in vector units. The second phase reads local scans and reductions from global memory to the vector cores where the scan computation is completed.

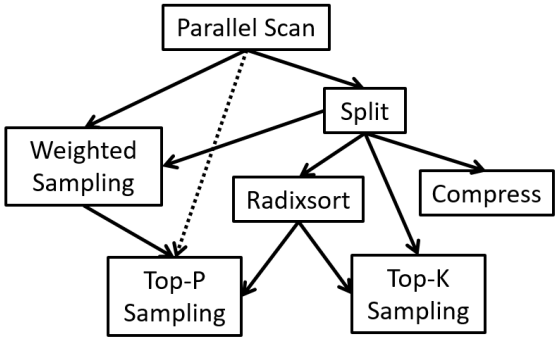

Fig. 7. A diagram of well-known parallel scan applications considered here along with their dependencies.

In the rest of the section, we describe the split, compress, radix sort, top-k, top-p sampling and weighted sampling operators in more detail. Figure 7 depicts the dependencies between the scan-based primitives. Top-p sampling extensively uses multiple scan operations, as discussed in Section 5.

Split. The split operation takes as input an array $\boldsymbol{x}$ and a boolean flag array $\boldsymbol{f}$ of the same length. Split reorganizes the elements of $\boldsymbol{x}$ into an output array $\boldsymbol{z}$ as follows. It places all items of $\boldsymbol{x}$ where the corresponding flag is true at the beginning of the output array, followed by all items where the corresponding flag is false. One crucial property of split is the relative order of the elements is preserved, i.e., *stable* ordering.

We implemented a split AscendC operator `SplitInd` that also returns the output indices corresponding to the original input locations. The output indices of `SplitInd` allow us to implement a sorting algorithm that satisfies the PyTorch API of `sort()`, which also returns the indices. `SplitInd` takes as input an array of 16-bit elements and a 0/1 mask array (flags are stored in int8). `SplitInd` 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. We have implemented a compress kernel that internally uses the exclusive `MCScan` algorithm on the mask array whose data type is 8-bit integers. Compress is equivalent to the PyTorch `torch.masked_select` operator that we use as a baseline for comparison in the experimental section.

Radix sort. Radix sort is a well-known application of the split operator [7, 9]. A radix sort algorithm loops over the bits of the input elements, starting at the least significant bit and executes a split where the mask is obtained by reading the corresponding bit (radix) on each iteration. 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 [24, Section 5.2.5]. The pre-processing phase encodes all the input elements by inverting the Most Significant Bit (MSB) of positive numbers and all the bits of the negative numbers. Applying an unsigned integer radix sort on the encoded elements will correctly order them. The post-processing phase is needed to decode the elements back to the original value. We have implemented the pre- and post-processing steps using AscendC bit-wise vector instructions, and thus, we support sorting of fp16 data types.

The paper on evaluating radix sort using the Connection Machine (CM-2) came to our attention in the later stages of our radix sort development [9]. In this work, the authors share several important implementation details that were not known to us a priori. For example, the fact that radix sort works with floats is interesting and is quite useful when working with low bit-width floats.

Top-k. Top-k selection is an essential operation in various settings, including similarity search queries [21] and Large Language Models (LLMs) inference where the output tokens are typically sampled from the k tokens having the highest probability for the given context [26]. The interested reader is referred to a recent survey on parallel top-k [49]. A recent work on top-k is Radik, a Radix-based GPU implementation that scales well for large values of k [29, 30].

We implemented a top-k kernel using the selection (partial quicksort) algorithm based on our `SplitInd` operator and compared it against the baseline top-k operator. Unfortunately, although improving the performance of the top-k operator was a primary motivation for this work, we could not improve the performance of the baseline top-k for small values of k ($k \leq 4096$).

Top-p or nucleus Sampling. Top-p sampling in Large Language Model inference is an additional operation that applies sort and scan on the token probability vector [16]. These operations are usually batched with a constant batch size. Interestingly, if the sorting step is implemented using radix sort, the top-p sampling operator becomes a scan-intensive operator! Indeed, top-p executes 17 scans for each batch: 16 scan operations for radix sort (one scan per bit, fp16) and an additional scan required by the algorithm.

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 [17].

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 [17] seems to be a promising direction.

6 Experimental Evaluation

In this section, we evaluate the performance of a selective list of parallel scan algorithms and applications presented in the previous sections using Ascend AI Accelerators.

We evaluate the multi-core scan, compress, radix sort, batch scan, and top-p sampling. We have implemented all proposed algorithms in C++17 using the AscendC programming framework discussed in Section 3.2. We used the Ascend CANN toolkit 8.0.RC3.alpha002 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 host CPU is an AMD EPYC processor running on Ubuntu 22.04.

All timing measurements are collected using the PyTorch profiler functionality in Python. We used Ascend's PyTorch adapter[2] with version v2.1.0 to report all our PyTorch-related measurements. To wrap our custom AscendC operators in PyTorch, we used the open-sourced operator plugin framework at https://gitee.com/ascend/op-plugin. `op-plugin` allows its users to easily define a custom PyTorch operator. Before wrapping our operators using the `op-plugin`, we used the `msopgen`[3] tool to wrap the AscendC operators.

Evaluation against other accelerators/architectures. In this work, our primary motivation is to improve the performance of parallel scans and their applications on Ascend. In particular, our main objective is to design and implement algorithms that saturate Ascend's memory bandwidth. Comparing our results with architectures like GPUs or TPUs would likely translate to comparing the hardware specifications, i.e. memory bandwidth, rather than actually evaluating our algorithms. In addition, a comparison between accelerators that are not manufactured using the same technology node could lead the reader to wrong conclusions. Nevertheless, we present all our results in terms of bandwidth (GB/s or GElems/s), which allows for easy comparison with other architectures for the interested reader.

6.1 Multi-core Scan (`MCScan`)

Figure 8 depicts the performance of `MCScan` (Algorithm 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

[2]https://gitee.com/ascend/pytorch
[3]https://www.hiascend.com/document/detail/en/canncommercial/700/operatordev/tbeaicpudevg/atlasopdev_10_0024.html

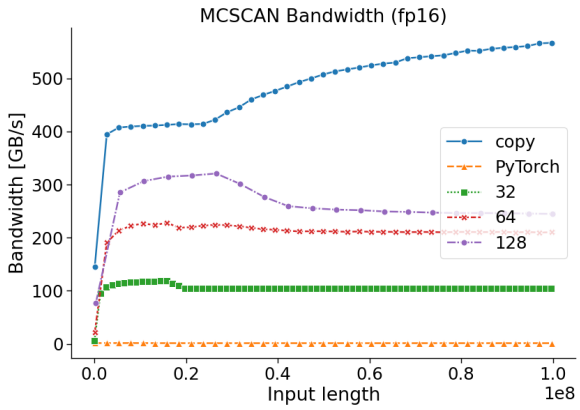

Fig. 8. Bandwidth of `MCScan` (Algorithm 3) for $s = 32, 64, 128$. In addition, the copy operator (`torch.clone`) is depicted for comparison. Memory bandwidth (910B4) is 800 GB/s.

and expose it as a custom PyTorch operator. The custom PyTorch operator statically pre-allocates an upper triangular all-ones matrix $\boldsymbol{U}_s$ for all $s = 32, 64, 128$. The baseline operator doesn't use the cube unit, while `MCScan` takes advantage of all the computing units reaching up to 37.5% of theoretical memory bandwidth (peak bandwidth is 800GB/s for 910B4). To have a more solid evaluation of our implementation, we compare it to a `copy` kernel that performs a memory copy; we used the `torch.clone()`. This shows us that for sizes smaller than the L2 cache, we almost approach the theoretical limit given by the memory bandwidth. A clear trend is that the larger the matrix multiplication dimension s is, the better the performance of the multi-core scan. $s = 128$ maximises the utilization of the level-0 scratchpad memories `L0A` and `L0B` of the cube unit. We have measured the speed-up between `MCScan` and `ScanUon` 910B4 (20 AI cores), and it saturates at 15.2× for large input sizes. We foresee that increasing the matrix multiplication tile size could lead to further performance improvements, but we leave this investigation as future work.

Next, we investigate the additional performance benefits of taking advantage of the lower-precision input data (int8) capability of the cube unit. Figure 9 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 it is depicted, there is a performance improvement of the order of 10% for the case of integer inputs. Such an improvement is crucial since the split and compress operators take as input boolean mask arrays that are stored[4] in int8 format.

6.2 Compress

Figure 10 depicts the performance comparison between compress versus the baseline PyTorch `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).

[4]This is due to the PyTorch framework, see for example https://github.com/pytorch/pytorch/issues/41571.

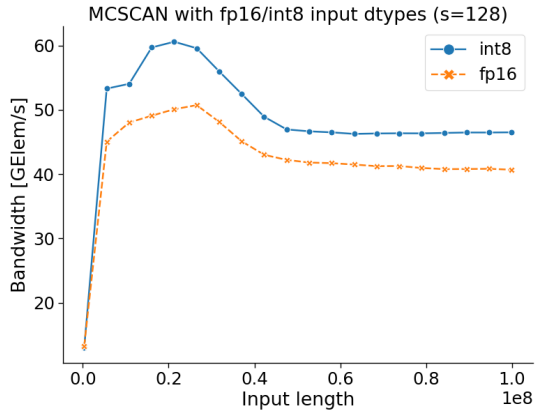

Fig. 9. Giga elements per second comparison of MCScan for float16 (fp16) and 8-bit integers (int8) input data types.

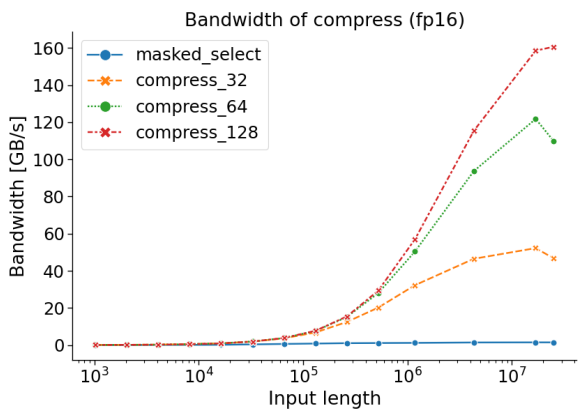

Fig. 10. Bandwidth of the compress operator based on MCScan ($s = 32, 64, 128$) and the baseline `torch.masked_select` operator.

6.3 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. The modification is based on the `SplitInd` operator by keeping track of the output indices on each split application. Our radix sort implementation is stable and supports unsigned (or signed) integers and floats (fp16).

Figure 11 depicts the performance of a parallel fp16 radix sort implementation using MCScan with input data type int8 (Algorithm 3) to perform the parallel split step. For input lengths greater than 525K, our "textbook" implementation of radix sort delivers a speedup between 1.3× up to 3.3× compared to the `torch.sort()` baseline.

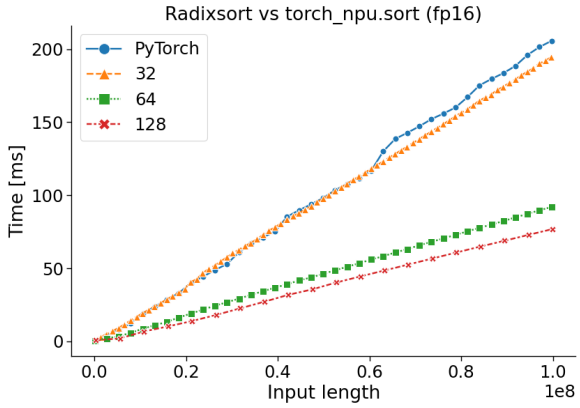

Fig. 11. Comparison between radix sort and `torch.sort()` for floating-point numbers in half-precision (16 bits).

We expect additional performance improvements 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 like 8-bit floats or 4-bit integers (int4) [40]. Therefore, an additional performance improvement (2×) for sorting in low-precision 8-bit scenarios is expected without further development effort.

6.4 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 enough, 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.

6.5 Top-p Sampling

Figure 13 depicts the execution time of drawing one sample using the top-p sampler as it is implemented in the Llama3 model, see [37]. 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. The figure shows that the baseline top-p sampling implementation scales poorly, mainly because the baseline `torch.cumsum` operator is not optimized for Ascend.

7 Conclusion & Future Work

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× to 9.6× compared to vector-only implementations for sufficiently large input lengths. Additionally, we presented a multi-core Ascend scan algorithm that fully utilizes both the cube and vector units of Ascend, reaching up to 37.5% of the theoretical memory bandwidth.

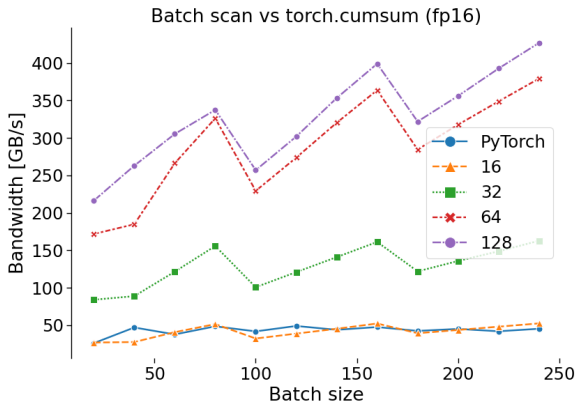

Fig. 12. Bandwidth of batched scan based on Algorithm 1 for increasing batch sizes and $s = 16, 32, 64, 128$. Input length is $65K$. Memory bandwidth (910B4) is 800 GB/s.

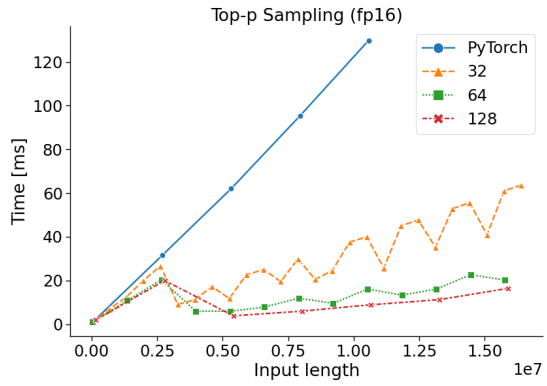

Fig. 13. Execution time (in milliseconds) of top-p sampling of Llama3 operator for a single batch. The baseline is labeled `PyTorch` and supports only discrete distributions with at most 2^{24} elements.

Last, we extended our contributions to include crucial computational kernels for AI workloads, such as parallel split, compress/compact, and top-p (nucleus) sampling, all exhibiting significant performance gains. Furthermore, our optimized implementation of radix sort, which utilizes matrix multiplications for parallel splits, showcases the potential of matrix engines in enhancing complex operations, offering up to 3.3× speedup over the baseline.

References

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

[2] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Munich, Germany) *(SPAA '08)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1378533.1378573

[3] ARM. 2024. ARM Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/ka/?lang=en

[4] Sean Baxter. 2016. moderngpu 2.0. (2016). https://github.com/moderngpu/moderngpu/wiki.

[5] Nathan Bell and Jared Hoberock. 2012. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann Publishers Inc., Boston, 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

[6] Guy E. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.* 38, 11 (1989), 1526–1538. https://doi.org/10.1109/12.42122

[7] Guy E Blelloch. 1990. Prefix Sums and Their Applications. In *Sythesis of parallel algorithms* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 35–60.

[8] Guy E. Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (March 1996), 85–97. https://doi.org/10.1145/227234.227246

[9] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Hilton Head, South Carolina, USA) *(SPAA '91)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/113379.113380

[10] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. 2020. A Computational Model for Tensor Core Units. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Virtual Event, USA). ACM, Philadelphia, USA, 519–521. https://doi.org/10.1145/3350755.3400252

[11] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. 2021. Algorithm Design for Tensor Units. In *International Conference on Parallel and Distributed Computing (Euro-Par)* (Lisbon, Portugal). Springer-Verlag, Lisbon, Portugal, 353–367.

[12] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing (ICS) (ICS '19)*. ACM, Phoenix AZ, 46–57. https://doi.org/10.1145/3330345.3331057

[13] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, and S. Matsuoka. 2021. Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?. In *International Conference on Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1056–1065. https://doi.org/10.1109/IPDPS49936.2021.00114

[14] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the ACM International Conference on Supercomputing (ICS)* (Island of Kos, Greece) *(ICS '08)*. ACM, New York, NY, USA, 205–213. https://doi.org/10.1145/1375527.1375559

[15] Albert Gu and Tri Dao. 2024. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. https://openreview.net/forum?id=AL1fq05o7H

[16] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, Addis Ababa, Ethiopia, 16 pages. https://openreview.net/forum?id=rygGQyrFvH

[17] Lorenz Hübschle-Schneider and Peter Sanders. 2022. Parallel Weighted Random Sampling. *ACM Trans. Math. Softw.* 48, 3, Article 29 (Sept. 2022), 40 pages. https://doi.org/10.1145/3549934

[18] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. 2023. Chapter 11 - Prefix Sum (Scan). In *Programming Massively Parallel Processors (Fourth Edition)* (fourth ed.). Morgan Kaufmann Publishers Inc., Cambridge, Massachusetts, United States, 253–256. https://doi.org/10.1016/B978-0-323-91231-0.00006-9

[19] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. 2023. *Programming Massively Parallel Processors (Fourth Edition)* (fourth ed.). Morgan Kaufmann Publishers Inc. 253–256 pages. https://doi.org/10.1016/B978-0-323-91231-0.00006-9

[20] Intel Corporation. 2022. *Intel Architecture Instruction Set Extensions Programming Reference.* Technical Report. Intel Corporation. https://www.intel.com/content/www/us/en/content-details/671368/intel-architecture-instruction-set-extensions-programming-reference.html Accessed: 2024-09-09.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[22] Norman P. Jouppi and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of International Symposium on Computer Architecture (ISCA)*. ACM, Toronto, ON, Canada, 1–12. https://doi.org/10.1145/3079856.3080246

[23] Norman P. Jouppi and et al. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i: Industrial Product. In *Proceedings of International Symposium on Computer Architecture (ISCA)*. IEEE, Online – Worldwide, 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[24] Donald E. Knuth. 1998. *The art of computer programming: sorting and searching* (2nd ed.). Vol. 3. Addison-Wesley Publishing Co., USA.

[25] Atsushi Koike and Kunihiko Sadakane. 2014. A Novel Computational Model for GPUs with Application to I/O Optimal Sorting Algorithms. In *International Conference on Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, Milan,

Italy, 614–623. https://doi.org/10.1109/IPDPSW.2014.72

[26] Wouter Kool, Herke Van Hoof, and Max Welling. 2019. Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement. In *International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 3499–3508. https://proceedings.mlr.press/v97/kool19a.html

[27] H.T. Kung and C.E. Leiserson. 1978. *Systolic Arrays for (VLSI)*. Carnegie-Mellon University, Department of Computer Science. https://books.google.ch/books?id=pAKfHAAACAAJ

[28] S. Lakshmivarahan and Sudarshan K. Dhall. 1994. *Parallel Computing Using the Prefix Problem*. Oxford University Press, Oxford, United Kingdom. https://books.google.ch/books?id=LQiRo-6GYdIC

[29] Yifei Li, Bole Zhou, Jiejing Zhang, Xuechao Wei, Yinghan Li, and Yingda Chen. 2024. POSTER: RadiK: Scalable Radix Top-K Selection on GPUs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Edinburgh, United Kingdom) *(PPoPP '24)*. ACM, New York, NY, USA, 472–474. https://doi.org/10.1145/3627535.3638478

[30] Yifei Li, Bole Zhou, Jiejing Zhang, Xuechao Wei, Yinghan Li, and Yingda Chen. 2024. RadiK: Scalable and Optimized GPU-Parallel Radix Top-K Selection. In *Proceedings of the ACM International Conference on Supercomputing (ICS)* (Kyoto, Japan) *(ICS '24)*. ACM, New York, NY, USA, 537–548. https://doi.org/10.1145/3650200.3656596

[31] X. Liang. 2020. *Ascend AI Processor Architecture and Programming: Principles and Applications of CANN*. Elsevier Science. https://books.google.ch/books?id=_bfjDwAAQBAJ

[32] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. 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)*. IEEE, Seoul, South Korea, 789–801. https://doi.org/10.1109/HPCA51647.2021.00071

[33] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *Hot Chips: A Symposium on High-Performance Chips (HCS)*. IEEE, Cupertino, CA, USA, 1–44. https://doi.org/10.1109/HOTCHIPS.2019.8875654

[34] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. In *MLSys*. mlsys.org, Santa Clara, California, USA, 1–14.

[35] Yossi Matias. 1997. Parallel algorithms column: on the search for suitable models. *SIGACT News* 28, 3 (Sept. 1997), 21–29. https://doi.org/10.1145/262301.262305

[36] Duane Merrill and Michael Garland. 2016. Single-pass Parallel Prefix Scan with Decoupled Lookback. In *Not available*. NVIDIA, Santa Clara, CA, USA, 1–9. https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

[37] Meta AI. 2024. LLama3 Generation Code - sample_top_p() method. https://github.com/meta-llama/llama3/blob/main/llama/generation.py#L358. Accessed: 2024-09-24.

[38] NVIDIA. 2023. Cooperative primitives for CUDA C++. https://github.com/NVIDIA/cub

[39] NVIDIA Authors. 2017. *NVIDIA DGX-1 With Tesla V100 System Architecture*. Technical Report MSU-CSE-06-2. Nvidia Corporation. 43 pages. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf

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

[41] Matt Pharr and Randima Fernando. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*. Addison-Wesley Publishing Co., Boston, USA.

[42] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *Proceedings of the Symposium on Graphics Hardware (EuroGraphics)* (San Diego, California) *(GH '07)*. Eurographics Association, Goslar, DEU, 97–106.

[43] Nodari Sitchinava and Volker Weichert. 2013. Provably Efficient GPU Algorithms. *CoRR* abs/1306.5076 (2013), 1–25. arXiv:1306.5076 http://arxiv.org/abs/1306.5076

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

[45] Lawrence Snyder, Leah H. Jamieson, Dennis B. Gannon, and Howard J. Siegel. 1985. *Algorithmically Specialized Parallel Computers*. Academic Press, Cambridge, Massachusetts, United States. https://books.google.ch/books?id=N9QmAAAAMAAJ

[46] William J. Starke, Brian W. Thompto, Jeff A. Stuecheli, and José E. Moreira. 2021. IBM's POWER10 Processor. *IEEE Micro* 41, 2 (2021), 7–14. https://doi.org/10.1109/MM.2021.3058632

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[48] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. *SIGPLAN Not.* 48, 8 (Feb. 2013), 229–238. https://doi.org/10.1145/2517327.2442539

[49] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. 2023. 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)* (Denver, CO, USA) *(SC '23)*. ACM, New York, NY, USA, Article 76, 13 pages. https://doi.org/10.1145/3581784.3607062

[50] Haikun Zhu, Chung-Kuan Cheng, and Ronald Graham. 2006. On the Construction of Zero-Deficiency Parallel Prefix Circuits with Minimum Depth. *ACM Trans. Des. Autom. Electron. Syst.* 11, 2 (April 2006), 387–409. https://doi.org/10.1145/1142155.1142162

[51] Anastasios Zouzias and William F. McColl. 2023. A Parallel Scan Algorithm in the Tensor Core Unit Model. In *International Conference on Parallel and Distributed Computing (Euro-Par) (Lecture Notes in Computer Science, Vol. 14100)*. Springer, Limassol, Cyprus, 489–502. https://doi.org/10.1007/978-3-031-39698-4_33

Received 20 February 20XX; revised 12 March 20XX; accepted 5 June 20XX