Efficient mmWave Point-Clouds for Embedded Devices

Evaluating Real-Time Performance of Embedded Millimeter-Wave Radar Pre-Processing Pipelines

Andreas Nicolaou

Supervisor(s): Marco Zuniga Zamalloa, Nicole Rosi

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2026

Name of the student: Andreas Nicolaou
Final project course: CSE3000 Research Project
Thesis committee: Marco Zuniga Zamalloa, Nicole Rosi, Jana Weber

Abstract

Automatically tracking the positioning and alignment of human limbs, also known as Human Pose Estimation (HPE), was traditionally pioneered by camera-based systems like the Microsoft Kinect, and remains critical across domains from interactive gaming to healthcare patient monitoring. Millimeter-wave (mmWave) radar has emerged as a compelling alternative; by utilizing electromagnetic waves to detect points on the surface of objects, it offers a more cost-effective, privacy-preserving, and robust solution than traditional cameras. However, the spatial "point-clouds" generated by mmWave radars are particularly irregular, requiring pre-processing before they can be fed into deep learning models. While these pre-processing techniques are well-documented and can easily be implemented on in high-level environments like Python, adapting and optimizing these pipelines for low-power embedded devices remains an underexplored challenge. It is currently not clear whether point-cloud pre-processing can overcome the memory and computational restrictions of low-power devices.

This thesis profiles the memory footprint and latency of executing mmWave point-cloud pre-processing on micro-controllers, specifically an **STM32 Cortex-M7 with 320 KB of SRAM** with the goal of real-time performance by processing each data sample in under 100 ms.

We propose and evaluate seven pipeline variants, incorporating hardware-acceleration, lightweight alternative algorithms, pipeline restructuring to eliminate computational redundancies, and a single-pass iteration strategy to minimize cache misses. Experimental results demonstrate that structural optimization compresses peak memory consumption from 90 KB to 50 KB, successfully approaching the theoretical lower bound dictated by the output buffers. Our most highly optimized configuration achieves an exceptional average latency of 8.13 ms (with a worst-case peak of 12 ms), comfortably satisfying our real-time constraints. Further analysis revealed that the average point count per frame is the primary driver of computational performance. Ultimately, this work validates that efficient, real-time end-to-end radar processing is entirely viable on highly resource-constrained micro-controllers.

1 Introduction

Human Pose Estimation (HPE) Systems, used to automatically track the position of movement of humans and their limbs, serve as the core engine behind interactive gaming, automated sports coaching, and smart fall-detection systems, contributing to fields from entertainment to healthcare. Historically, HPE has relied on optical camera systems like the Kinect, but while cameras provide rich visual data, they introduce severe privacy concerns when deployed in both private and public spaces and may fail entirely in poor lighting conditions or with a foggy lens.

To overcome these limitations, millimeter-wave (**mmWave**) radar has emerged as a compelling alternative. By utlising high-frequency electromagnetic waves, mmWave radars detect points on the surfaces of objects without capturing any identifiable visual details or depending on light-conditions. It provides a cost-effective, more private, and environmentally robust solution.

However, mmWave radar data is fundamentally different from the clean grid of camera pixels. Theradar front-end outputs data as a collection of "point-clouds", 3D points on the surfaces of objects, per time-frame. Unlike other radar technologies like LiDAR, mmWave point-clouds are highly irregular, unevenly spaced, and a different number of them are captured at every time-frame. Therefore, before we can use standard deep learning methods on point-clouds to learn and infer human pose, we must process them into fixed-dimension input-ready data. This pre-processing typically includes transformation, normalization, and up- and down-sampling.

HPE models and pre-processing techniques are well researched and can run with great precision on powerful desktop systems [1; 2; 3], but often their ultimate goal is to be deployed on small embedded devices. While model compression is a thoroughly explored subject [4; 5], there is no current profile of how point-cloud pre-processing performs on low-power devices. To assess the efficiency of these techniques and identify potential bottlenecks, we propose and evaluate a total of 7 variants of a pre-processing pipeline, incorporating hardware acceleration, lightweight alternative algorithms, pipeline restructuring to eliminate computational redundancies, and a single-pass iteration strategy to minimize cache misses. All tested variants are functionally equivalent, that is, they all have the same output for every input. We evaluate these implementations on an **STM32** micro-controller (MCU) (**Cortex-M7**) equipped with **320 KB** of **SRAM**, using data samples from the MARS dataset [6].

The primary research question driving this study is: **How efficiently can mmWave radar point-cloud pre-processing pipelines be executed on memory-constrained micro-controllers, and can they meet real-time performance requirements?**

The main question can be broken down into the following sub-questions:

1. **Memory Constraints;** What is the lowest achievable worst-case peak memory usage of the end-to-end processing pipeline when executed on the STM32 MCU, and can it fit within the 320 KB SRAM constraint?
2. **Latency Profiling;** What is the lowest achievable worst-case latency of the pipeline when executed on the micro-controller?

3. **Real-Time Integration;** Can the combined pre-processing and inference pipeline meet real-time constraints when both are executed on the MCU?

Real-time constraints We aim for an end-to-end latency of under 100 ms for real-time inference, therefore we set a strict pre-processing latency limit of 100 ms. An algorithm that runs under this limit with the **worst-case input** may qualify for real-time use.

The remainder of this paper is structured as follows: Section 2 provides background information on mmWave radar systems, point-cloud representations and processing requirements for deep learning. Section 3 defines the research problem and presents an overview of the pre-processing pipeline and the target embedded platform. Section 4 describes the different implementation and optimization approaches used. Section 5 outlines the profiling methodology used to evaluate latency and memory usage, followed by Section 6 which presents and analyzes the experimental results. Section 7 reflects on the ethical aspects and the reproducibility of the chosen methodology. Finally, Section 8 concludes the paper and outlines directions for future research.

2 Background & Related Work

2.1 A review of mmWave FMCW Radars

Millimeter-wave (mmWave) Frequency Modulated Continuous Wave (FMCW) radar systems have emerged as an important sensing modality for human detection, tracking, and pose estimation tasks. Operating in the 30-300 GHz frequency range, mmWave radars estimate the distance, velocity, and angular position of objects using reflected electromagnetic waves. These characteristics make mmWave radars particularly attractive for indoor and outdoor monitoring, healthcare and human pose estimation applications.

The generation and interpretation of mmWave radar point clouds have been extensively reviewed by Harry D. Mafukidze et al. [7] who describe the complete mathematical transformation from radar samples to point clouds and how they can be prepared for feature extraction and classification. Traditional radars analyze each reflected signal they receive, forming sparse points containing information on the motion, geometry, and physical properties of objects, estimated using complex signal processing techniques. However, the short wavelength of mmWave radars produces too many reflections from complex object surfaces, so analyzing all of them individually is not practical. Instead, clusters of these detections are represented by point-clouds to avoid complex and redundant computations. A mmWave point-cloud consists of 3-D cartesian coordinates, the Doppler velocity, and the signal intensity.

2.2 Point-Cloud feature extraction

Unlike LIDAR point-clouds, mmWave point-clouds are irregular and not uniformly distributed in space; that is, there is no consistent definition for how many points are detected per time frame, or the distance between 2 neighboring points [7]. Hence, feature extraction from a point-cloud data stream can be challenging. Two primary approaches exist for converting point clouds into deep learning inputs: Point clouds can be *processed* into normalized, fixed-dimension representations with extracted features that are directly compatible with specialized architectures such as PointNet [8]. Alternatively, they can be *transformed* into special data structures, enabling the use of standard deep learning techniques that are not specifically designed for point-cloud data.

Processing Some models can use point cloud data along with some basic geometric or statistical features. Recent examples are Zhao et al [9], who developed a model which uses 11 such features extracted from raw point-clouds, Jin et al [3], who developed a fall-detection model using the extracted centroid of point-cloud clusters, and Xue et al [1] who developed an HPE model utilizing range, the distance from a point to the origin.

Transformation Other models utilize voxelization or conversion of point-clouds to 2D images. These methods produce uniform, well-defined data that can be used in standard deep learning operations. Examples include the voxelization algorithm proposed by Wu et al [10] who achieved high accuracy for posture classification. These algorithms can be very resource-demanding, especially with memory, and are therefore not the ideal choice for deploying on resource-constrained devices.

In this paper, we are interested in **Point-Cloud Processing**.

2.3 Point-Cloud Pre-Processing for Neural Networks

Recent advances in radar signal processing and deep learning have enabled mmWave systems to perform increasingly sophisticated perception tasks. Examples include the mmFall model designed by Jin et al [3] which uses a Hybrid Variational Recurrent Neural Network Autoencoder (HVRAE) to detect human falls, and mmMesh by Xue et al [1], designed to reconstruct a 3D mesh of a person's body using mmWave point-clouds, by incorporating the PointNet architecture [8]. Although these two systems use different network architectures, they have very similar pre-processing needs that are relevant to this project:

Radar Mounting Position Changing the radar mounting positions will offset the Euclidean space in which point-clouds are placed. Both architectures require that input data accounts for the mounted position of the radar, by transforming the point-cloud's cartesian coordinates accordingly.

Up-sampling Neural networks have fixed-dimension inputs, therefore only accept data of the same size. For HPE models, this means that their input consists of a fixed number of time-frames, each with a fixed number of point-clouds. When the radar has not collected enough points for a time-frame, we must up-sample it, i.e. populate it until it has the correct amount of points. Typical up-sampling techniques include zero padding, which just appends zeroes to the frame and random sampling, which duplicates random points from the frame with noise. Jin et al proposed a novel

data up-sampling algorithm for mmFall, which is designed to keep the point-cloud mean and variance constant as their model relies on geometrical centroids. The MARS dataset [6] used zero padding to up-sample their frames, and in this report we will be using **random sampling**.

Downsampling Similarly, we must ensure time-frames don't contain too many points, and if they do we must clip them. Xue et al generate a consistent number of point-clouds per time-frame for mmMesh, by capturing raw radar information, and selecting reflections with the highest signal intensities. Choosing points with the highest intensities also has the advantage of eliminating noise and artifacts from the multi-path effect [1, pg. 275]. In this report, we utilize data from the MARS dataset [6] captured from the radar front-end, therefore we cannot use the same operations, instead we emulate them by sorting and clipping point-clouds.

Order invariance Different permutations of spatial data are equivalent, therefore models must be invariant to input order. Qi et al [8] go into detail on how making a model invariant to unordered inputs is a non-trivial task, as stable sorting for 3-D points is not possible. The PointNet architecture uses the max pooling operation as a symmetric function to aggregate information from point clouds, regardless of order. mmMesh on the other hand, uses the attention mechanism [1] which is also symmetric, to avoid potential information loss from the max pooling function.

Feature extraction Both mmFall and mmMesh extract geometrical features from their point-cloud inputs: mmFall extracts the centroids of point-cloud clusters, to estimate the center of the body of each person and detect sudden vertical movements, and mmMesh calculates the range, the distance from the radar to each point, for each point-cloud. Both architectures use these features as an additional dimension to the network inputs.

3 Problem Definition & System Overview

In this section, we define the problem by detailing the pre-processing requirements mentioned in Section 2.3, and providing an overview of the target platform. We begin with the detailed input requirements of the relevant model, before analyzing all stages of the pre-processing pipeline illustrated in Figure 1. In Section 3.3, we provide the hardware specifications of the target platform and finally in Section 3.3 we describe the low-level memory configuration used.

3.1 AI Model & Input Requirements

The pre-processing requirements used in this study are based on the mmWave radar human pose estimation system developed by Yukuan et al. [2], which builds upon the mmMesh architecture [1] and PointNet [8]. The model takes an input tensor of 32 time frames, each containing 64 points described by six features: 3D Cartesian coordinates, Doppler velocity, signal intensity, and range (the distance from the origin).

3.2 Pipeline Structure & Derivation

This section outlines the stages of the baseline pipeline. Figure 1 illustrates the complete end-to-end pipeline along with brief descriptions of each stage. As reviewed in Section 2, the following sequential processes are required to condition the raw data into consistent, fixed-dimension inputs suitable for neural networks:

1. **Time Sequence Clipping:** Extracting the most recent t time frames captured by the radar.
2. **Coordinate Transformation:** Standardizing the data across varying radar mounting positions by translating and rotating all point clouds accordingly.
3. **Up-sampling:** Ensuring a uniform number of points per time frame by populating frames that fall below the target threshold. The up-sampling approach implemented here selects a random point from the frame, duplicates it, and applies Gaussian noise to the clone's Cartesian coordinates. This process is repeated until the target count is reached.
4. **Down-sampling:** Truncating data from frames that exceed the target number of points. To preserve the most significant data and eliminate noise, points are selected based on the highest signal intensities [1]. This is achieved by sorting the points in descending order of signal intensity and retaining the first x_{target} points.
5. **Normalization:** Normalizing the Doppler and signal intensity values of each point using the standard formula: $\frac{x-\mu}{\sigma}$ where μ (mean) and σ (standard deviation) are constant parameters derived from the dataset.
6. **Compute Range:** The mmMesh network [1], which our model is based on, requires that each point is paired with its range (i.e. its Euclidean distance from the origin) for correct inference. This value is computed independently for each point using its Cartesian coordinates via: $\sqrt{x^2 + y^2 + z^2}$.

These processes yield an output tensor of size $t \times x_{\text{target}} \times 6$, representing t time frames containing x_{target} points, with each point characterized by 6 dimensions. As mentioned in Section 3.1, this study uses $t = 32$ and $x_{\text{target}} = 64$.

3.3 System Overview

System Specifications

The pipeline implementations are evaluated on an STM32 development board featuring the following relevant hardware specifications [11]:

1. A Cortex-M7 processor operating at 200 MHz.
2. A Floating Point Unit (FPU) providing hardware acceleration for the conversion, addition, subtraction, and multiplication of single-precision (32-bit) floating-point numbers.
3. Two general-purpose, dual-port Direct Memory Access (DMA) controllers supporting memory-to-memory transfers.

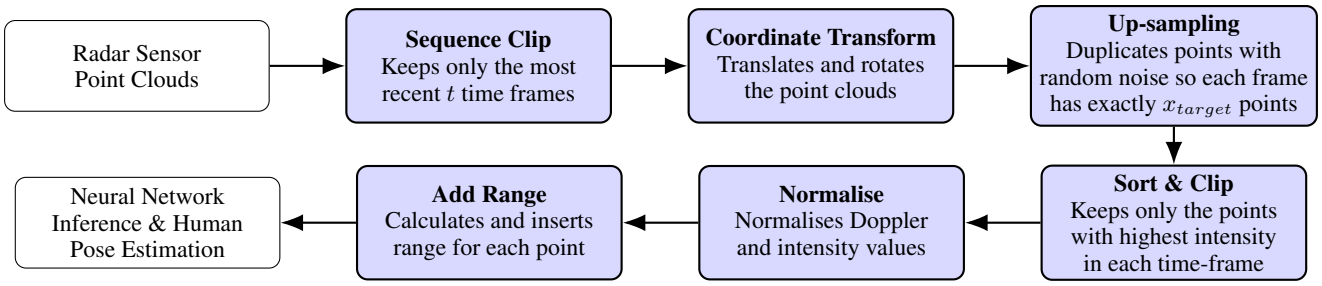


Figure 1: End-to-end mmWave radar human pose estimation pipeline. Highlighted stages represent the pre-processing pipeline originally implemented in Python for testing and development, and later implemented and evaluated on the STM32 micro-controller.

4. A True Random Number Generator peripheral.
5. System SRAM of up to 320 Kbytes, where memory segments are mapped starting from the Data Tightly-Coupled Memory (DTCM) at the lowest address:
   (a) SRAM1 on the AHB bus matrix: 240 Kbytes
   (b) SRAM2 on the AHB bus matrix: 16 Kbytes
   (c) DTCM-RAM via the TCM interface: 64 Kbytes
6. Flash memory of up to 1 Mbyte.

The DTCM, unlike the standard SRAM, is attached directly to the processor, therefore offering zero-delay access times. It is also not cached, as it is essentially as fast as the caches themselves.

The software implementations evaluated in this paper are written in C, leveraging standard libraries (such as `stdlib.h` and `math.h`) alongside the STM32 Hardware Abstraction Layer (HAL) drivers. Compilation is performed using the *gnu-tools-for-stm32*[1] toolchain (version `14.3.1+st.2`) with the following optimization flags enabled:

1. `-O3`
2. `-mfloat-abi=hard` (Enables FPU utilization)

Memory Mapping

As noted in Section 3.3, the lowest 64 KB of the system RAM address space are mapped to the DTCM, which offers zero-delay, uncached memory access. The linker script used for **all test configurations** organizes the memory layout from lowest to highest address as follows:

1. Initialized data segment (`.data`)
2. Uninitialized data segment (`.bss`)
3. Heap base (allocating upwards)
4. Stack pointer base (allocated at the highest RAM address, growing downwards)

The executable code segment (`.text`) is stored entirely in Flash memory. This conventional layout places the stack within the slower segments of SRAM, while the heap begins allocations within the faster DTCM. Because our application's stack depth never exceeds 256 KB during execution, the stack resides completely within the slower SRAM. Using the heap would pose two issues; (A) If the heap allocates a buffer partially outside of the DTCM and in the slower SRAM, it would have inconsistent access times, and (B) parts of the buffer inside the DTCM would completely skip caching. To ensure our profiling results are consistent and applicable to embedded applications constrained to cached or non-TCM memory regions, we strictly avoided heap allocations and instead allocated all data buffers on the stack to rigorously evaluate the impact of caching on the performance of data pre-processing on embedded devices.

[1] https://github.com/STMicroelectronics/gnu-tools-for-stm32

4 Embedded Implementation

This section outlines the key implementation choices and optimization techniques employed throughout the project, building upon the pipeline operations detailed in Section 3. First, we analyzed the most computationally intensive operation, up-sampling, and identified the optimization techniques detailed in Section 4.1 aimed at mitigating latency overhead. Subsequently, we pinpointed redundant operations within the pipeline and eliminated them by introducing the restructured flow described in Section 4.2. Finally, recognizing that the baseline pipeline design yields suboptimal cache utilization, we developed the single-pass approach explained in Section 4.3.

4.1 Optimizing Up-sampling

Random up-sampling, detailed in Section 3.2, tends to be an intensive process in most contexts, as it generates a vast number of random numbers. When run on embedded systems, it presents another concern; as the output of up-sampling contains more values than its input, it cannot work in-place, and must instead transfer all data to an output buffer. These random number generations and memory transfers can become very expensive on resource-constrained devices.

Random Number Generation

A primary performance concern with random up-sampling is the computational efficiency of random number generation (RNG). The up-sampling method used in this study works by choosing a random point from the frame, duplicating it, and adding some gaussian noise to the duplicate's cartesian coordinates, then repeat this procedure until the frame is sufficiently populated. This requires two distinct RNG streams: a **uniform distribution** to select which points to

duplicate, and a **normal (Gaussian) distribution** to apply noise to the coordinates of the duplicated points.

Gaussian distribution To implement noise generation efficiently, we leverage the **Box-Muller transform** [12], which generates pairs of independent, normally distributed random numbers from a source of uniformly distributed inputs.

Uniform distribution While the standard C library function `rand()`[2] provides a built-in pseudo-random number generator for uniform samples, it introduces non-trivial overhead. While using `rand()` as the baseline, we also test the board's built-in **TRNG peripheral**, which uses an analog live entropy source to provide true random numbers. As a more lightweight alternative, we also evaluate the **XORShift** algorithm proposed by Marsaglia [13]. XORShift requires only three bitwise XOR and shift operations to generate pseudo-random sequences of 32-bit numbers.

Parallelizing Data Transfers

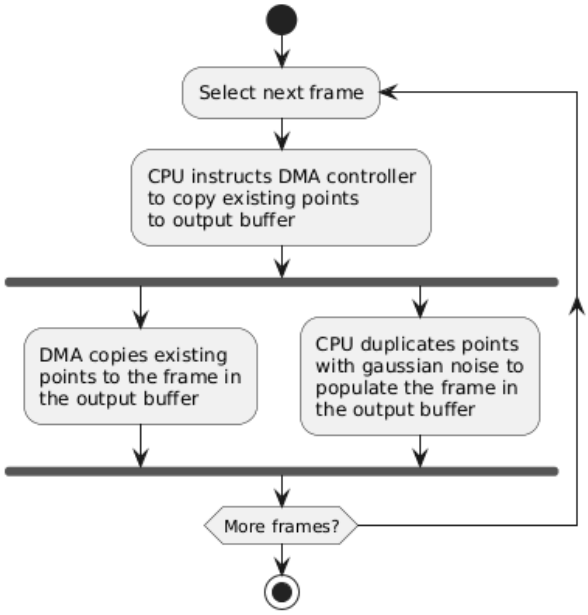


Figure 2: Activity diagram for memory transfer parallelization using the DMA-Controller when up-sampling time-frames. The DMA copies existing points to the new buffer while the CPU deals with generating new points.

During the up-sampling phase, the input buffer only stores the original point-clouds, and therefore is too small to accommodate both the original and the synthesized data. Therefore, we create a new output buffer designed to accommodate the full up-sampled data. Existing point clouds must be transferred to this expanded output buffer, which requires simply copying all existing point clouds within each frame directly to their corresponding frame in the output buffer. Because this data is transferred without modification, the memory copy operation can be executed in parallel with point generation.

On the STM32 architecture, the internal Direct Memory Access (DMA) controller (mentioned in Section 3.3) can offload these memory copy tasks, freeing the CPU to simultaneously handle up-sampling and Gaussian noise generation. To do this, we implement the execution model shown in Figure 2: for each time-frame, the CPU instructs the DMA to copy the frame's existing data, and immediately begins generating the synthetic points needed to populate the remainder of that same frame. After completing the generation, the CPU waits until the DMA transfer finishes before proceeding to the next frame.

4.2 Re-structuring the Pipeline

After discovering several redundant computations and memory allocations in the original pipeline, we applied the techniques described in this section to eliminate them, minimizing both latency and memory footprint. This section details the pipeline variant referred to as **Restructured Flow** in the following sections. This variant is illustrated in Figure 3.

Elimination of Redundant Operations

Clip first, process later The `Sequence Clip` stage obtains the first 32 frames of a sample, while the `Sort & Clip` stage clips frames to 64 points each, by deleting excess points. As there is no reason to process points that will be removed in the next stage, we move all processing stages downstream of the `Sort & Clip` stage, so they only process points that will not be removed. Similarly, the `Sort & Clip` stage remains after the `Sequence Clip` stage so it doesn't sort frames that will be later removed.

Conditional Sorting The `Sort & Clip` function only needs to sort point in order to retain the 64 points with the highest signal intensity, but if there are less than 64 points in a frame, there is no reason to sort them. As mentioned in Section 2.3, point-cloud-specialized neural networks are invariant to the order of point-clouds. Therefore, there is no need to sort the points in a time-frame unless it contains more than 64 points.

Process first, duplicate later The `Up-sampling` stage populates frames by duplicating points. When a point is duplicated, its Doppler velocity and signal intensity remain unchanged. Coincidentally, the `Normalization` stage only processes these two values. Therefore, we place the `Normalization` stage upstream of `Up-sampling`. In this configuration, only the original points will go through normalization, and then get duplicated, leading to the same output. This greatly reduces the number of points that go through normalization. Note that the `Add Range` stage cannot be similarly relocated as the point duplication alters the Cartesian coordinates used to calculate the range, which would require the range to be re-calculated for each duplicated point.

Reducing memory buffers

The `Up-sampling` and `Add Range` stages each require an output buffer larger than their respective inputs, as the former

[2]https://pubs.opengroup.org/onlinepubs/7908799/xsh/rand.html

increases the number of points, and the latter increases the number of features of each point. Hence, both stages must instantiate and populate independent output buffers. While the last instantiated buffer is used as the output buffer, the other one is temporary and is discarded after use. This temporary buffer not only massively raises memory usage, but the associated memory copy operations greatly increase latency.

To minimize memory overhead, we introduce the `Copy Buffer` operation shown in red in Figure 3. This intermediate step allocates a single, terminal output buffer to be used by both subsequent stages and copies the input data into it. The succeeding up-sampling and range-computation stages then execute their tasks in-place without spawning additional memory structures or performing redundant copy operations.

4.3 Single-pass: Improving cache utilization

Previous variants operate on a stage-by-stage basis, iterating over the whole sample once for each stage. Each stage starts on the first frame of the sample, which has likely been replaced in the cache by another frame after the previous stage's iteration. This results in a cache miss, costing valuable wait-state time.

Instead, we propose the single-pass approach, building on the restructured flow to prioritize spatial locality; by processing data frame-by-frame and point-by-point, we ensure that each piece of memory (frame or point) is operated on with a single batch operation. This increases memory access predictability and minimizes cache invalidation, reducing latency.

Algorithm 1 outlines the pseudocode for this approach. The main logic is as follows: For each frame, points are sorted and clipped if necessary, then transformed and normalized as they are transferred directly to the output buffer. Immediately after individual point processing, the frame is up-sampled to the target point density, and the range for each point is computed. This localized execution strategy aims to maximize data locality and minimize cache misses by completing all operations consecutively on each memory segment.

Algorithm 1 Single-pass algorithm

```
Ensure: ||frames|| = 32
  for all frame in frames do
    if ||frame|| > 64 then
      frame ← sort_and_clip(frame)
    end if
    for all point in frame do
      new_buffer ← normalize(transform(point))
    end for
    new_buffer ← up_sample(new_buffer)
    for all point in new_buffer do
      new_buffer ← add_range(point)
    end for
  end for
```

5 Profiling and Evaluation Methodology

This section outlines the methods and datasets used to verify and profile the tested pipeline variations. We start by explaining the correctness testing methods, before detailing the dataset source and sampling techniques used. Finally, we outline the data collection methods used to profile the two metrics evaluated in this paper; peak memory footprint and latency.

5.1 Correctness

To test and debug the code, we used UART to send the output of each stage from the micro-controller to a computer running a python script. The python script captured the output, and compared it with outputs from the original python pipeline code. This was not possible for non-deterministic operations, such as up-sampling, so we used Kolmogorov-Smirnov tests [14] to ensure that the noise added to the samples followed a normal distribution at the 5% significance level.

5.2 Input Data

Data samples from the MARS Dataset [6] were used to test and evaluate the pipeline variants. To perform a valid experiment, it's important to compare different software under the same conditions, but it is equally as important to test each variant under different circumstances to explore how different inputs affect the performance of the software. To apply this, we strategically chose 12 samples from the dataset to represent a stratified collection of circumstances. We chose samples that reflect extreme test-cases of the following statistical features. The sample numbers used are listed in the appendix, in Table 2:

1. Average Number of point-clouds per time-frame
2. Variance in number of point-clouds per time-frame

5.3 Memory Usage

As mentioned in Section 3.3, we decided against using any heap-allocated memory to test the impact of memory caching. Since the output buffer size is constant for each configuration, it is possible to statically allocate it, completely avoiding dynamic memory allocation for point-cloud pre-processing. Therefore, stack usage is a sufficient metric for evaluating the memory efficiency of different pipeline variants.

Memory usage was profiled using the system tick handler, which is called in 1 ms intervals. The handler keeps track of the minimum value of the stack pointer (as the stack grows downwards, this represents its maximum size), using a global variable. After execution, we subtract the minimum recorded stack pointer with the stack pointer recorded just before pipeline execution, to obtain the maximum stack usage.

5.4 Latency

Latency was calculated using the Data Watchpoint and Trace (DWT) Unit's cycle counter, part of the hardware debugging extension of the Cortex M-7 processor. The system clock is configured at the start of the program at 200MHz, and the cycle count is copied to memory after every pipeline stage. Using the system clock speed and the cycle count at each

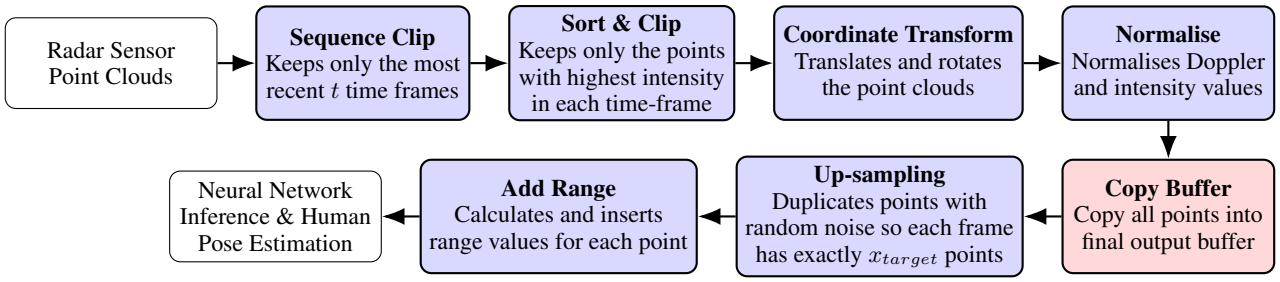


Figure 3: Restructured end-to-end mmWave radar human pose estimation pipeline. Blue-highlighted stages represent the pre-processing pipeline originally implemented in Python for development and testing, and later implemented and evaluated on the STM32 micro-controller, while red-highlighted stages are introduced in this report for optimization on embedded devices.

stage we can reliably and accurately measure the latency in milliseconds.

One notable consideration is the impact of the overridden system tick callback function used to log the stack memory usage, mentioned in Section 5.3. This operation would typically add around 13-20 CPU instructions per millisecond, resulting in a negligible worst-case delay of 100 nanoseconds per millisecond.

6 Results & Analysis

We evaluated seven pipeline configurations to assess performance and resource utilization. Implementation details for the approaches used in these variants are provided in Section 4:

- **Variant 1**: Original Pipeline, FPU disabled.
- **Variant 2**: Original Pipeline, FPU enabled (Baseline).
- **Variant 3**: Original Pipeline, FPU, TRNG-based RNG.
- **Variant 4**: Original Pipeline, FPU, XORShift-based RNG.
- **Variant 5**: Original Pipeline, FPU, XORShift RNG, DMA-accelerated memory transfers.
- **Variant 6**: Restructured Flow, FPU, XORShift RNG.
- **Variant 7**: Restructured Flow, FPU, XORShift RNG, **Single-Pass** implementation of the restructured flow.

Variant 2 (FPU-Enabled) is designated as the baseline, as it represents a standard deployment configuration. The non-FPU variant serves only as a reference point and is not denoted as the baseline because its latency is approximately an order of magnitude higher than all other variants, making comparisons less meaningful. Table 1 summarizes the average latency across all samples for main variants.

6.1 Memory Usage

Memory consumption remained consistent within each pipeline architecture: the original structure required approximately **90 KB**, while the new structure reduced this to about **50 KB**. Stack usage is occupied primarily by output buffers; based on the parameters in Section 3, the final output buffer requires 49,152,Bytes. The original pipeline's higher memory usage stems from the intermediate buffer used between up-sampling and range computation. As all variants remain well within the 320 KB memory limit, we conclude that all configurations are feasible for micro-controller deployment.

6.2 Latency

Figure 4 illustrates the latency trends across all variants. The FPU-disabled variant (Variant 1) averaged **93.6 ms** (max **130.7 ms**). This does not satisfy the 100 ms real-time constraint, indicating a lack of viability for pre-processing on hardware without floating-point units. Enabling the FPU (Variant 2) reduced latency to **15.8 ms**, a significant improvement showing that the FPU is a critical performance factor, enabling real-time usage.

Regarding RNG methods, the on-board TRNG (Variant 3) was slower than the standard C `rand()` function used in the baseline variant, averaging **16.5 ms**. Conversely, the lightweight XORShift algorithm (Variant 4) outperformed both at **14.4 ms**. Integrating DMA memory parallelization (Variant 5) further reduced latency to **14.0 ms**. While the performance increase is slight, it validates the use of DMA to optimize up-sampling in embedded environments.

The proposed new pipeline (Variant 6) significantly outperformed both the baseline and the XORShift+DMA variant (Variant 5), reaching **9.6 ms**. As shown in Figure 5, this is largely due to the elimination of redundant sorting. This flow also improves the efficiency of normalization, range computation, and up-sampling, at the cost of the negligible "Copy Buffer" stage. Finally, the single-pass implementation (Variant 7) achieved the peak performance of **8.1 ms**, with a worst-case latency of **12.0 ms**. By consolidating individual stages into batch operations, this variant greatly reduces the latency of the transformation, normalization, and add range stages, likely by reducing cache misses, hence minimizing total execution time.

6.3 Analysis of Sample Characteristics

Figure 4 illustrates the total latency across all variants, while Figure 6 breaks down the execution time by individual stages. We observe that samples with less points per frame exhibit significantly higher total latency, with the up-sampling stage consuming the majority of execution time. Conversely, samples with higher point densities require approximately

Variant	Peak Stack Usage (KB)	Average Latency (ms)	Maximum Latency(ms)
Without FPU	90.63	93.64	130.34
With Hardware FPU (Baseline)	90.71	15.75	21.49
With XORShift & DMA	90.74	13.98	19.0
Restructured Flow	50.08	9.62	14.28
Single-Pass	50.15	8.13	12.04

Table 1: Performance evaluation for the main pipeline variants, gathered over 12 samples

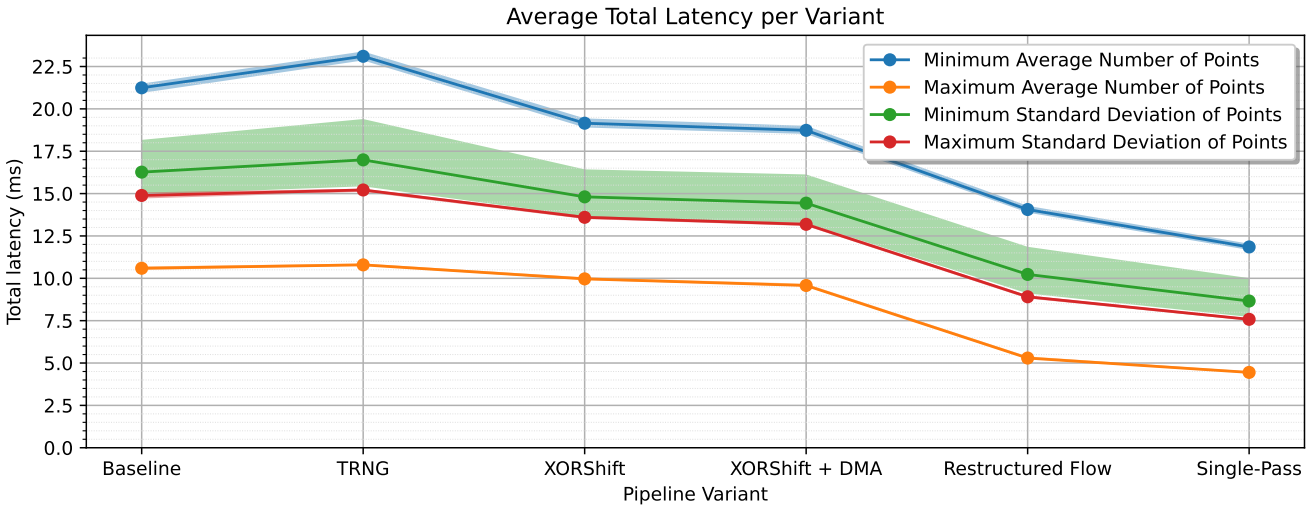


Figure 4: Average latency for each pipeline variant, for 4 groups of 3 dataset samples each. The error areas are bound by the minimum and maximum recorded latency. The statistical significance of each sample group is denoted in the legend.

half the processing time, primarily because the up-sampling stage scales down proportionally. The less points present in a time-frame, the more points must be generated, greatly increasing the amount of computation required.

The two variance-based sample groups exhibit average latencies relative to the extreme groups. While the minimum standard deviation group displays higher latency variability, this is entirely due to the inconsistency in the average number of points within those specific samples rather than the variance itself. These results indicate that the absolute average number of points per frame is the dominant factor determining latency, whereas the consistency of point counts across frames has a negligible impact.

7 Responsible Research

This section addresses the ethical implications of mmWave radar technology and evaluates the reproducibility of the empirical results presented in this report.

7.1 Ethical Concerns

While advancing data pre-processing on its own introduces minimal ethical concerns, the broader implications of advancing mmWave radar technology require careful consideration. As established previously, mmWave radars offer superior privacy preservation compared to camera-based sensors because they omit visual imaging. However, other properties of mmWave technology still present significant privacy risks:

Through-wall sensing Millimeter-wave signals can penetrate obstacles such as glass, wood, and drywall. Phaisiri et al. [15] demonstrated that these signals can successfully detect and track human activity through concrete and glass barriers.

Audio Eavesdropping Due to their very high resolution, mmWave radars can detect microscopic surface vibrations, enabling them to capture and reconstruct audio broadcasted from speakers. Pengfei et al. developed milliEar [16], a system that leverages this phenomenon alongside machine learning to extract and reconstruct intelligible audio.

These capabilities underscore that while mmWave radars avoid visual identification, they still present privacy and ethical concerns. By profiling and optimizing real-time pre-processing on embedded devices, hence increasing the efficiency and accessibility of this technology, we implicitly lower the barrier to its deployment. Consequently, widespread adoption must be accompanied by rigorous deployment safeguards, transparency measures, and ethical guidelines to mitigate potential misuse and safeguard individual privacy.

7.2 Integrity and Reproducibility of Results

Latency As outlined in Section 5, latency was measured via the microprocessor's internal cycle counter and computed based on a fixed 200 MHz clock frequency. Because the pipeline operates without interrupts or external peripherals, execution remains highly deterministic, yielding reliable latency metrics.

Memory Usage Memory usage was monitored by sampling the stack pointer every millisecond. The stack pointer's value can change at any time in just a single cycle, and if this change is reverted in less than a millisecond, it could potentially go unnoticed. Furthermore, due to the deterministic nature of the system, sampling would *always* occur at the same point in the program execution, so running the code multiple time would yield the same result.

One potential solution we considered was to use a randomized delay at the start of the program, to offset the system clock and induce non-deterministic sampling. However, this was not necessary for two reasons; (A) The memory footprint is completely dominated by the output buffers, which are statically allocated and predictably sized, and (B) the results obtained were highly consistent within pipeline structures and samples, making us confident about our sampling methods.

Although the code used for these experiments is not publicly available as of publishing this thesis, researchers should be able to replicate the reported results by closely following the implementation details in Section 4, the detailed compilation and hardware configurations specified in Section 3.3, and even by using the exact samples used, listed in Table 2.

7.3 Limitations

One limitation of the dataset used is that all frames in every sample contain ≤ 64 points. Therefore, when implementing the conditional Sort & Clip stage described in 4.2, which skips sorting for frames with ≤ 64 points, we effectively completely disable the sorting stage for all tested samples. While this does not necessarily affect the inferences made from results, it is important to consider the implications; testing variants with frames that exceed 64 points should show the same latency and memory usage trends, but the restructured variants of the pipeline may present higher average latency than reported as some frames will get sorted.

8 Conclusions and Future Work

This work demonstrates that an mmWave point-cloud preprocessing pipeline can successfully execute within a micro-controller environment, requiring as little as **50 KB** of peak memory, approaching the theoretical limit imposed by the output buffer. Our evaluation reveals that a DMA controller is a viable mechanism for accelerating up-sampling memory transfers. Furthermore, restructuring the pipeline to eliminate redundancies (Variant 6) reduces both latency and memory footprint, while executing batch operations (single-pass, Variant 7) yields greater efficiency than independent stages. Regarding random number generation, the on-board TRNG module proved slower than the standard C `rand()` function, whereas the lightweight XORShift algorithm[13] outperformed both. Utilizing the revised pipeline flow, a single-pass approach, and the XORShift algorithm allows the fastest variant to achieve a worst-case latency of just **12 ms**, safely satisfying the constraints for real-time inference.

Future work will focus on integrating the downstream machine learning model directly onto the micro-controller and deploying a physical mmWave radar sensor to test and profile the complete, end-to-end human pose estimation pipeline in real-time.

A Further analysis of latency data

This appendix section consists of figures that are relevant to results analysis but not necessarily to answer the main research questions. Figures 5 and 6 break down the latency of each stage for further analysis of computational bottlenecks. Figure 5 shows the total average latency data for 2 variants, while Figure 6 shows latency data for each sample group, to identify circumstantial bottlenecks.

B Samples used

Table 2 lists the sample numbers used to obtain the results for this report, as imported from the MARS dataset [6].

References

[1] H. Xue, Y. Ju, C. Miao, Y. Wang, S. Wang, A. Zhang, and L. Su, "mmmesh: Towards 3d real-time dynamic human mesh construction using millimeter-wave," in *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, pp. 269–282, 2021.

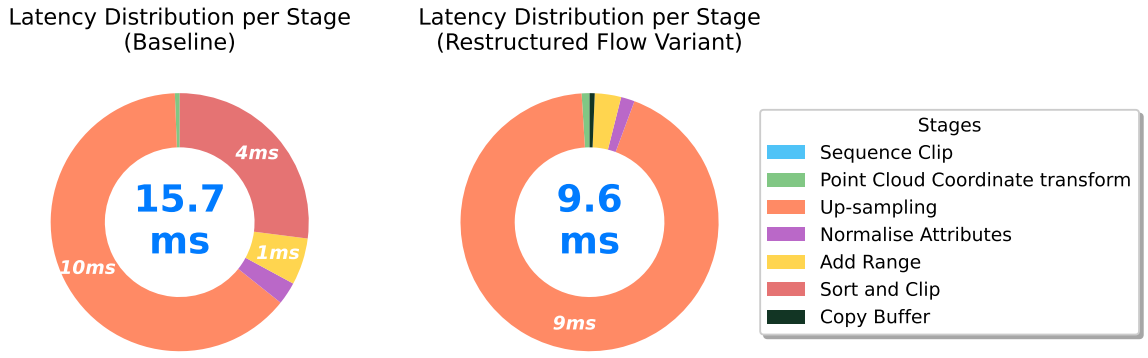
[2] Y. Ding, H. Martinez, G. Vaidya, K. Langendoen, and M. Z. Zamalloa, "Towards device-free gaming with mmwave radar," in *2026 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 1–11, 2026.

[3] F. Jin, A. Sengupta, and S. Cao, "mmfall: Fall detection using 4-d mmwave radar and a hybrid variational rnn autoencoder," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 2, pp. 1245–1257, 2020.

[4] H. A. Abushahla, D. Varam, A. J. N. Panopio, and M. I. AlHajri, "Neural network quantization for microcontrollers: A comprehensive survey of methods, platforms, and applications," 2026.

[5] I. Lamaakal, C. Yahyati, I. Ouahbi, K. E. Makkaoui, and Y. Maleh, "A survey of model compression techniques for tinyml applications," in *2025 International Conference on Circuit, Systems and Communication (ICCSC)*, pp. 1–6, 2025.

[6] S. An and U. Y. Ogras, "Mars: mmwave-based assistive rehabilitation system for smart healthcare," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–22, 2021.



(a) Baseline (b) New Flow

Figure 5: Latency distribution per Stage for 2 variants; Baseline and Restructured Flow

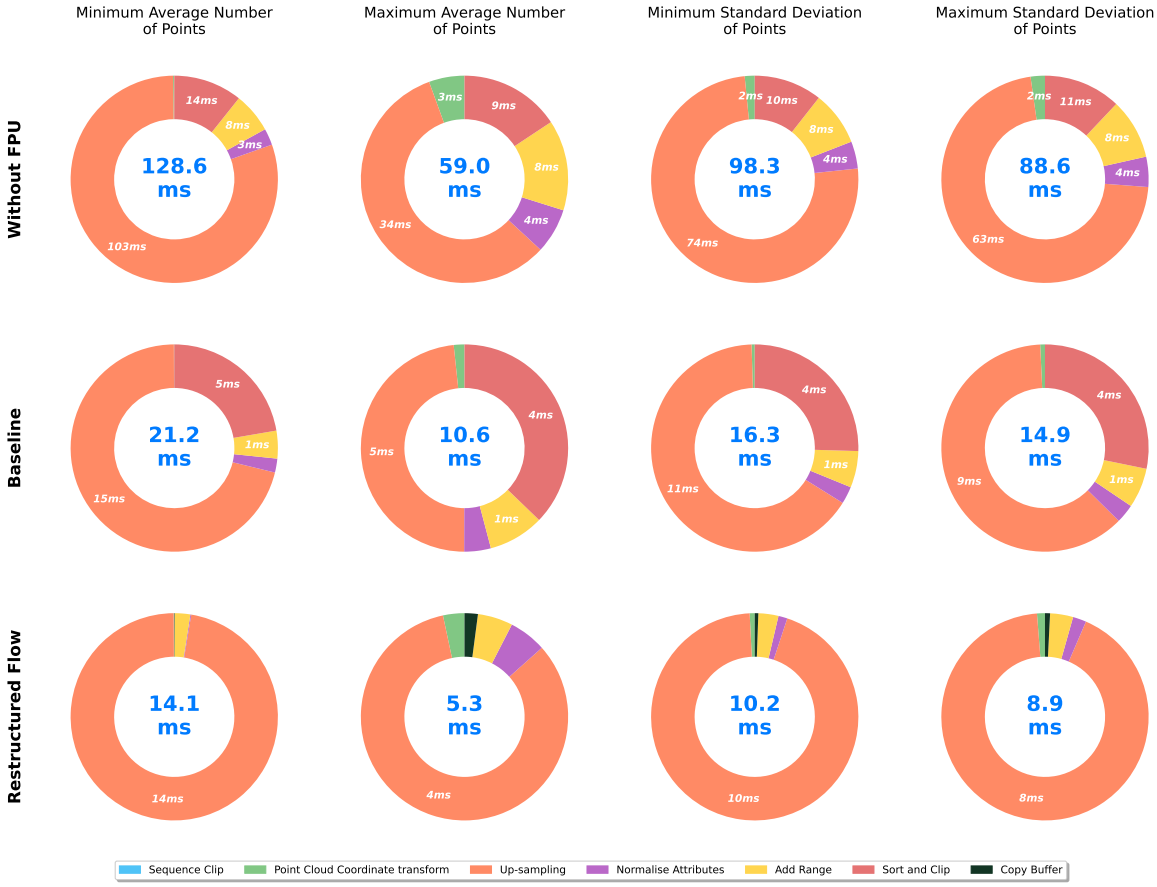


Figure 6: Breakdown of latency distribution per stage, for each variant and each sample group, denoted by the group's statistical characteristic. The only variants displayed are ones with notable differences in stage latency distribution.

Sample group	Statistical Significance	Sample 1	Sample 2	Sample 3
1	Minimum Average Number of Points per Frame	0	1	2
2	Maximum Average Number of Points per Frame	1331	1332	1333
3	Minimum Variance of number of Points per Frame	1817	1703	44
4	Maximum Variance of number of Points per Frame	662	663	664

Table 2: List of sample groups and sample numbers used, taken from the MARS Dataset

[7] H. D. Mafukidze, A. K. Mishra, J. Pidanic, and S. W. Francois, "Scattering centers to point clouds: A review of mmwave radars for non-radar-engineers," *IEEE Access*, vol. 10, pp. 110992–111021, 2022.

[8] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2017.

[9] Z. Zhao, Y. Song, F. Cui, J. Zhu, C. Song, Z. Xu, and K. Ding, "Point cloud features-based kernel svm for human-vehicle classification in millimeter wave radar," *IEEE Access*, vol. 8, pp. 26012–26021, 2020.

[10] J. Wu, H. Cui, and N. Dahnoun, "A voxelization algorithm for reconstructing mmwave radar point cloud and an application on posture classification for low energy consumption platform," *Sustainability*, vol. 15, no. 4, p. 3342, 2023.

[11] STMicroelectronics, "STM32F745XX, STM32F746XX Datasheet - Production Data," 7 2025.

[12] G. E. Box and M. E. Muller, "A note on the generation of random normal deviates," *The annals of mathematical statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[13] G. Marsaglia, "Xorshift rngs," *Journal of Statistical software*, vol. 8, pp. 1–6, 2003.

[14] V. W. Berger and Y. Zhou, *Kolmogorov-Smirnov Test: Overview*. John Wiley & Sons, Ltd, 2014.

[15] K. Phaisiri, A. Boonpoonja, C. Kittiyanpunya, K. Kaemarungsi, and N. Chudpooti, "Through-wall human sensing using mmwave fmcw radar," in *2023 Research, Invention, and Innovation Congress: Innovative Electricals and Electronics (RI2C)*, pp. 1–4, 2023.

[16] P. Hu, W. Li, Y. Ma, P. S. Santhalingam, P. Pathak, H. Li, H. Zhang, G. Zhang, X. Cheng, and P. Mohapatra, "Towards unconstrained vocabulary eavesdropping with mmwave radar using gan," *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 941–954, 2024.