Investigating Array Programming for Spatial Operations with Vector Geometries

Balthasar Teuscher[1], Xuanshu Luo[1], Paul Walther[1], Martin Werner[1]

[1] Technical University of Munich, Germany; TUM School of Engineering and Design, Department of Aerospace and Geodesy, Professorship of Big Geospatial Data Management - (balthasar.teuscher, xuanshu.luo, paul.walther, martin.werner)@tum.de

Keywords: Array Programming, Vector Geometry, Data Structures, Vectorization.

Abstract

Vector geometries are traditionally represented as entities of sequences of coordinate structures. With advancements in hardware and software for data analytics, a preference for columnar data layouts arose. This paper examines array programming for spatial operations to evaluate the potential performance benefits of modern computing architectures and emerging spatial data encodings. Evaluating selected operations, such as distance calculation, extent extraction, and affine transformations, indicates similar or improved performance for geometries with columnar coordinate layouts. By leveraging modern compiler infrastructure, we further demonstrate that advanced hardware features in commodity hardware, such as vector instructions, are becoming accessible without specialized code. The performance comparison with established, widely used geospatial and computational libraries reveals significant untapped potential for increasing the efficiency of spatial computing.

1. Introduction

The ever-increasing volume of spatial data collected challenges the performance of analytical methods and infrastructure in efficiently generating insights. In response to this challenge, specialized designs such as neural processor units and architectural extensions featuring new instructions have emerged to accelerate data-intensive computing. However, effective use is rarely a given, as software generally targets a conservative feature set, leaving untapped the potential for optimization.

Processing collections of records often involves applying the same operation to each value of an attribute. Array programming and single instruction, multiple data (SIMD) allow operations to be applied to a set of values simultaneously. SIMD originated from a classification of very high-speed computing systems (Flynn, 1966). It refers to a type of processor that can exploit data parallelism. Over the years, various architectures and instruction sets have emerged, increasing their availability even on commodity hardware. Unfortunately, leveraging these features requires specialized programming language skills and code that many high-level languages lack. However, modern compilers such as LLVM are capable of translating and lowering code to utilize these instructions without the need for any adaptations to a certain extent through loop unrolling and automatic vectorization[1].

In the geospatial domain, a persistent drawback is the dominant geometry encoding model, which relies on entities with sequences of coordinate structures, that is ill-suited for array programming. Furthermore, the tooling is embedded within the broader context of specific programming language environments and dependencies, inheriting their benefits and drawbacks, including those of the entire computing stack built upon them.

This paper examines the potential of modern hardware developments, such as AVX, and software advancements, including SIMD and LLVM, for accelerating spatial computations. The focus lies on evaluating array programming for operations on vector geometries. The main contributions are a thorough review of vector geometry representations, examples of applying array programming to spatial operations, and a comparative performance evaluation of selected instruction sets and computational libraries.

[1] `https://llvm.org/docs/`, last accessed 30.03.2026

2. Related Work

High-performance and accelerated computing is an ongoing research area since the inception of the computer and is often considered to evolve exponentially (Kurzweil, 2001). Most prominent recent examples include the emergence of Neural Processing Units (NPUs) designed to accelerate the growing Machine Learning (ML) and Artificial Intelligence (AI) workloads (Esmaeilzadeh et al., 2012). Similar trends occur in the evolution of the Central Processing Unit (CPU), such as increases in core counts, wider vector units, and the introduction of processing arrays to increase parallelization of data-intensive computations (Kristof et al., 2012).

In this regard, single instruction, multiple data (SIMD) has a long history and is widely explored, especially for analytical workloads and query execution in databases (Polychroniou et al., 2015, Habich and Pietrzyk, 2024). Likewise, it has been explored in the geospatial domain, such as accelerating indices (Rayhan and Aref, 2023) or columnar data formats (Saeedan and Eldawy, 2022). Still, efficient use of specific hardware features often entails substantial implementation overhead, specialized knowledge, and imposes specific hardware requirements.

The ongoing specialization and fragmentation of hardware and software have led to the development of abstraction layers that, together with intermediate representations and compiler infrastructure, provide access to heterogeneous hardware functionality through a unified API and without specialized code, thereby increasing portability tremendously (Anderson et al., 2016, Lattner et al., 2021).

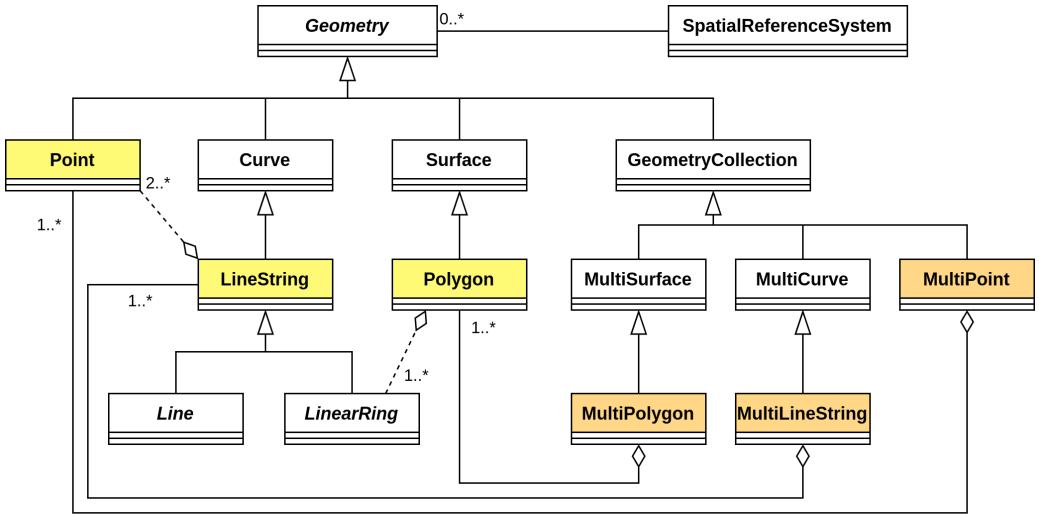

Figure 1. The conceptual vector geometry model defined by OGC Simple Features in a UML diagram notation.

3. Background

This section outlines some fundamental concepts of spatial computing, including vector geometry representation, hardware acceleration, and compiler infrastructures.

3.1 Geometry Representation

Representing geometries for spatial computing is versatile, driven by domain-specific needs and applications. In the following, we provide a brief overview of foundational aspects of the most common geospatial vector geometry representation, including its conceptual model, standard encodings, underlying data structures, and related access patterns.

Conceptual Model. A somewhat canonical representation of geographic features is formalized in the Simple Features standard for two-dimensional geometries (OGC/ISO, 2011). It includes a conceptual model for geometries consisting of an abstract *Geometry*, concrete types and subtypes thereof, and how they relate to, respectively, are composed of each other (compare Figure 1). In practice, the model is often simplified to only consider the *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString*, *MultiPolygon*, and *GeometryCollection* types as in the GeoJSON specification (Butler et al., 2016).

Encodings. Various encodings of geometries from the above conceptual model exist. Well-known text (WKT) and GeoJSON define textual encodings for geometries, while well-known binary (WKB) and variations thereof specify binary encodings for geometries (OGC/ISO, 2011, Butler et al., 2016). In programming languages and libraries such as GEOS/JTS and `liblwgeom` from PostGIS[2], geometries are often modeled relying on their native type system. A newer encoding is GeoArrow[3]. It is based on Apache Arrow[4], a columnar in-memory format optimized for vectorized execution.

Data Structures. Coordinates are commonly represented as tuples or structures with a numeric scalar for each dimension. Points encompass one set of coordinates, whereas a line string is commonly modeled as an array of points, and a polygon as an array of line strings. Multiparts are simply single parts encapsulated in a list. The abstract *Geometry* is often modeled as a sum type, or an enumeration of the concrete geometry types. Using tuples for coordinates and arrays for multiple values is considered an array of structs (AoS) data structure. Alternatively, coordinates can be represented in a single interleaved array, or as a structure of arrays (SoA) where coordinates for each dimension are stored in separate arrays. The latter has been prototyped in the GeoArrow native encoding with separated coordinates[5].

Memory Layout. To efficiently use SIMD, the array of scalars should be aligned to a multiple of the instruction size in adjacent memory. Unfortunately, this is not enforced by default for arrays and containers in most programming languages. To address this, Apache Arrow[6] specifies a memory layout where buffers are aligned, recommending a multiple of 64 bytes for effective AVX-512 utilization.

Access Patterns Computing with geometries requires accessing their coordinates. While any possible access pattern can be emulated and abstracted over the underlying data structure and layout, their performance implications depend heavily on how well-suited these abstractions are to compiler optimizations, available intrinsics, and code lowering, among other factors.

3.2 Harware Acceleration

Hardware is the physical infrastructure where computation actually takes place. As such, it is at the other end from conceptual models and theoretical algorithms. To translate abstract problems into executable solutions through concrete implementations, we outline general concepts of hardware acceleration that enable high-performance computing, focusing on parallelization and its interactions with data storage and movement.

[2] `https://github.com/postgis/postgis/blob/master/liblwgeom/gserialized.txt`, last accessed 30.03.2026
[3] `https://geoarrow.org/format.html`, last accessed 30.03.2026
[4] `https://arrow.apache.org/docs/format/Columnar.html`, last accessed 30.03.2026
[5] `https://geoarrow.org/format.html`, last accessed 30.03.2026
[6] `https://arrow.apache.org/docs/format/Columnar.html`, last accessed 30.03.2026

Vector Instructions One of the lowest interfaces for a programmer to access is the set of instructions for a given processor architecture. These core computational capabilities and primitives are often backed by dedicated hardware for execution. Single instruction, multiple data (SIMD) vector instructions allow an operation to be performed on a set of values simultaneously in parallel on a single core. The Advanced Vector Extensions (AVX) for x86[7], the Neon instruction set for Arm[8], and the RISC-V Vector extension[9] being the most prominent today. With AVX-512, input arguments and outputs of intrinsics can be 512 bits, which is equivalent to 8 64-bit floating-point values. For example, when adding two arrays element-wise, a chunk of 8 elements from each array is loaded into the registers, added, and the result is stored, requiring a total of 4 instructions. This is theoretically 8 times faster. However, it is difficult to achieve and sustain, as data cannot be moved from memory at this speed, and a complete program often requires additional instructions not available. Still, even partial use can lead to substantial overall performance improvements.

Multiprocessing & -threading Distributing computational workloads among multiple processing cores is another common approach for parallelization. While multiprocessing allows for true isolated parallelism, which means running several tasks in parallel as individual processes, starting these tasks usually implies additional overhead for the operating system. Further, tasks are memory-isolated, with each assigned a separate exclusive portion of memory. Multithreading, on the other hand, is a lightweight alternative with lower overhead, less isolation, and shared memory access. Another approach is used on graphics processing units (GPUs), consisting of more but simpler cores that run the same program on different chunks of data. Parallelizing programs is not always trivial, and even when it is, how you split the problem with respect to data locality needs to be considered to achieve cache efficiency.

Storage & Fusion Storage plays a crucial role in accelerated computing. It is commonly conceptualized in a hierarchical, layered fashion, from persistent storage with large capacity and slow throughput (e.g., a hard drive), through volatile memory with medium throughput, capacity, and random access, to cache (e.g., shared L3 or per-core L1) close to the central processing unit with small capacity and high throughput, and finally registers. Moving data through these layers is complex, though necessary for efficient, effective processing. Two key aspects in this regard are locality and size, as data is often managed in chunks of a given capacity, which are cached. With good data locality and adherence to capacity, caching becomes effective, and data can reside at higher levels of the storage hierarchy for reuse, reducing the need to move data from slower lower levels to the processing unit and thereby reducing computation stalls. This also hints at doing as much processing as possible on the data available. An approach for this is kernel fusion, where consecutive operations are executed on a slice of data, and only the final output is written back to memory, whereas in eager execution, the intermediary output of each operation is written back to memory.

[7] `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html`, last accessed 30.03.2026

[8] `https://www.arm.com/technologies/neon`, last accessed 30.03.2026

[9] `https://lists.riscv.org/g/tech-vector-ext/attachment/691/0/riscv-v-spec-1.0.pdf`, last accessed 30.03.2036

3.3 Compiler Infrastructures

Programming languages require an interpreter or a compiler to translate code into a representation that the hardware can execute. One such compiler infrastructure is LLVM.

LLVM The technologies of LLVM are a modular set of tools that allow for applying various optimizations. One such optimization is loop unrolling and autovectorization[10]. Through this, high-level programming code can be translated into a form that potentially uses vector extensions of the target hardware without explicit annotations. The modularity of LLVM makes it extensible and enables it to target a wide range of platforms and processor architectures.

MLIR Multi-Level Intermediate Representation[11] (MLIR) is a sub-project of LLVM that provides an extensible intermediate representation (IR), which allows accessing heterogeneous hardware-specific acceleration (Lattner et al., 2021). It is intended to provide greater flexibility and enhanced capabilities, such as parallelizing code across cores.

4. Methodology

To investigate array programming for spatial operations with vector geometries, we implemented a selection of basic spatial operations on top of two containers for the coordinates, once an array of structs (AoS) as a list of tuples, and once a struct of arrays (SoA) as a tuple of lists, and finally benchmark them by compiling and running with various instruction set extensions.

4.1 Spatial Operations

The kernel selection is intended to cover computational problems, such as element-wise distance and affine transformation, as well as aggregations such as centroid and extent.

Distance. Measurement functions between points are fundamental spatial operations. Most prominent is the Euclidean distance d between two points or arrays of points $\mathrm{p}_1, \mathrm{p}_2$ given by:

$$\mathrm{d} = \sqrt{(\mathrm{p}_1[1] - \mathrm{p}_2[1])^2 + \cdots + (\mathrm{p}_1[m] - \mathrm{p}_2[m])^2}, \quad (1)$$

where $\mathrm{p}_{\ldots}[m]$ denotes the coordinates of the m-th dimension. This direct, simple distance calculation is sometimes replaced with the $hypot(x, y)$ function for improved numeric stability, at the cost of performance. Additionally, when the distance is solely used for ordering, the square root can be omitted to optimize.

Affine. Transforming coordinates is a common operation, for example, translating projected coordinates to image coordinates when rasterizing and during reprojections or transformations between coordinate reference systems in general. Basic transformations, like scale, rotation, and shift, can be achieved by an affine transformation, which can be expressed with a homogeneous affine matrix A in 2D as:

$$A = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \quad (2)$$

[10] `https://llvm.org/docs/Vectorizers.html`, last accessed 30.03.2036

[11] `https://mlir.llvm.org/`, last accessed 30.03.2036

The transformed points p' are then obtained as

$$\begin{aligned} \mathrm{p}'[1] &= a_{11}\mathrm{p}[1] + a_{12}\mathrm{p}[2] + b_1 \\ \mathrm{p}'[2] &= a_{21}\mathrm{p}[1] + a_{22}\mathrm{p}[2] + b_2. \end{aligned} \tag{3}$$

Centroid. Calculating the centroid c is a geometry processing operation where the arithmetic mean of a set of n points is computed. This can be expressed for dimension m as:

$$c_m = 1/n \sum_{n=1}^{n} p_n[m]. \tag{4}$$

Occasionally, a weighted approach is used to reflect the geometry's area or length in the weight.

Extent. The extent E of a geometry is the minimum and maximum coordinate value per dimension. Formally expressed as the interval for each dimension m:

$$E_m = [\min_{1 \le i \le n}(\mathrm{p}_i[m]), \max_{1 \le i \le n}(\mathrm{p}_i[m])], \tag{5}$$

where p is the array, respectively, the set of n points of the geometry.

4.2 Implementation

We implement two naive versions of the above spatial operations. The AoS version is a simple list of coordinate tuples, whereas the SoA version is a tuple of lists. For calculating the distance, affine and extent, the kernels remain the same, and only the access pattern differs: iterating over tuples compared to creating them by zipping two iterators. The centroid was calculated by summing the tuples for AoS, compared to accumulating the sum over each array for SoA, and then dividing by the number of points. Using the same kernels isolates the impact of the data layout, thereby improving their evaluation. Further, we use Rust, a compiled systems programming language that uses LLVM as its compiler backend. Additionally, we included Burn[12], a tensor library similar to PyTorch, that offers autotuning, kernel fusion, various GPU backends, and an experimental CPU backend based on MLIR. Finally, we implement the same operations with GeoPandas, Numpy, Geo (Rust)[13], Apache Arrow (Rust)[14], and on GPU and CPU using Burn to extend and contextualize the experiments.

4.3 Benchmarking

To evaluate the impact of different approaches, we establish benchmarks in a stable environment to ensure consistent results. For the microbenchmarks, we use criterion[15], a dedicated benchmarking library. This yields more stable, reproducible results by including a 3-second warmup phase, followed by about 100 samples. Further, by using a compiled language such as Rust, we can explicitly select the target features for compilation and execution. This allows us to enable or disable certain instruction sets for comparison.

[12] `https://github.com/tracel-ai/burn`, last accessed 30.03.2036
[13] `https://github.com/georust/geo`, last accessed 30.03.2036
[14] `https://github.com/apache/arrow-rs`, last accessed 30.03.2036
[15] `https://github.com/criterion-rs/criterion.rs`, last accessed 30.03.2036

5. Experimental Results

In this section, we present the experimental results of our investigation into array programming with vector geometries. It involves comparing data layouts, instruction sets, and various computational libraries.

5.1 Experimental Setup

All the following experiments were conducted on a mobile computer equipped with an AMD Ryzen™ 7 7840U processor featuring 8 cores (16 threads) and 32 GB of LPDDR5 (6400 MT/s) memory. The processor is of the Zen 4 generation and supports advanced vector extensions up to AVX-512F (AVX-512 Foundation); however, the actual hardware implementation for the latter is two 256-bit consecutive vectors. The GPU benchmarks were running on the internal AMD Radeon™ 780M. As input, 300'000 points were generated.

5.2 Instruction Sets and Layouts

To evaluate different data structures and corresponding memory layouts for coordinates in combination with various vectorization instruction sets, we implemented and benchmarked the spatial operations introduced in Section 4.1, once as a simple list of tuples (AoS), a tuple of lists (SoA), an version of the latter with explicit SIMD instruction and finally based on Apache Arrow (Table 1). It should be noted that the explicit SIMD implementation is not well developed and follows a naive approach, relying on the portable SIMD capabilities of nightly Rust. A highly optimized implementation is represented by Arrow.

Operation	ISA	AoS	SoA	SIMD	Arrow
		µs	µs	µs	µs
	SSE	305	301	321	303
	AVX	286	144	174	146
Distance	AVX2	145	142	162	144
	AVX512	136	133	152	135
	native	130	129	148	129
		µs	µs	µs	µs
	SSE	158	105	207	139
	AVX	134	96	136	135
Affine	AVX2	124	100	133	136
	AVX512	104	107	179	144
	native	102	92	176	129
		µs	µs	µs	µs
	SSE	192	87	92	97
	AVX	186	82	67	52
Centroid	AVX2	184	82	65	51
	AVX512	190	86	67	42
	native	186	87	69	41
		µs	µs	µs	µs
	SSE	221	209	225	206
	AVX	116	67	75	70
Extent	AVX2	97	67	70	69
	AVX512	68	54	101	56
	native	66	52	106	52

Table 1. Comparison of the array of structures (AoS) and structure of arrays (SoA) data layout for spatial operations compiled and executed with various instruction set extensions.

The instruction sets were passed to the compiler by means of the *target-feature* variable in an additive manner from SSE with *target-feature=-avx,-avx2,-avx512f*, equal to the default or no flags up to $AVX512$ as *target-feature=+avx,+avx2,+avx512f*. Additionally, $native$ corresponds to the *target-cpu=native*

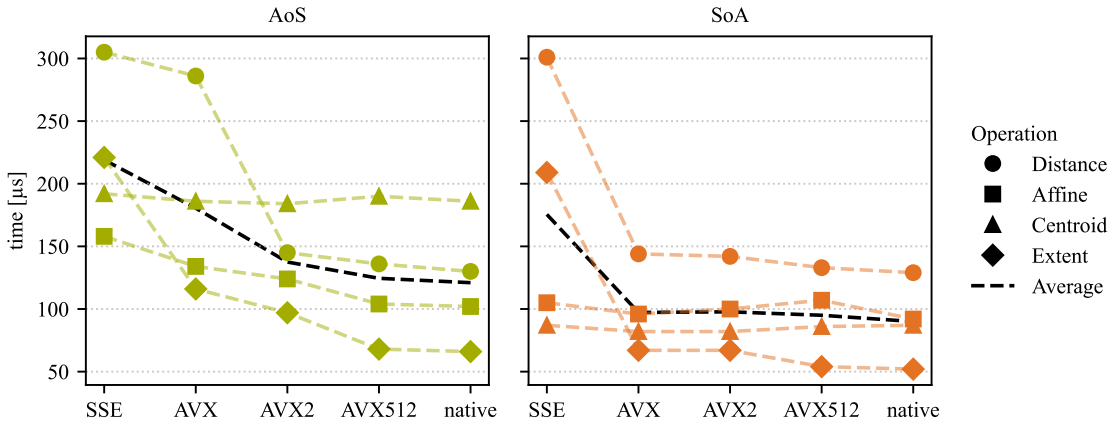

Figure 2. Runtime comparison of data layouts and spatial operations compiled and executed against different instruction set extensions.

compiler option, which activates all available features. The results show that vectorization can speed up certain kernels up to 4 times, and that SoA memory layout is up to twice as fast without any regressions compared to AoS. The greatest gain results from SoA with vector extension, whereas the width is of lesser (Figure 2). Additionally, the custom naive SIMD implementation displays mixed results, whereas Arrow shows more stable performance similar to SoA.

5.3 Computational and Geospatial Libraries

To assess the potential benefit of array programming, we compare our naive implementation with native compilation against GeoPandas and the underlying NumPy library, which is specialized for array programming and vectorized execution, once with a C layout (SoA) and once with a Fortran layout (AoS). Additionally, we compare against Geo, a computational geometry library, and Arrow, both written in Rust, as well as against Burn using the MLIR-based CPU and WebGPU backends (Table 2).

	Distance	Affine	Centroid	Extent
	ms	ms	ms	ms
GeoPandas	7.48	3494.04	3.56	5.67
NumPy (AoS)	2.82	0.72	3.60	12.15
NumPy (SoA)	2.23	0.35	0.09	0.10
Geo (Rust)	1.14	0.96	0.31	0.23
Arrow (Rust)	0.13	0.13	**0.04**	**0.05**
Burn [cpu]	0.23	0.25	0.63	0.76
Burn [gpu]	0.67	0.74	0.61	0.75
Naive (AoS)	0.13	0.11	0.18	0.11
Naive (SoA)	**0.12**	**0.09**	0.08	**0.05**

Table 2. Comparison of our approach to selected spatial operations in common libraries.

The GeoPandas library tends to exhibit some overhead, most notably for affine transformations, which, as a basis for coordinate transformations, shows great potential for performance improvements. Most interestingly, the premise that SoA is equal to or superior to AoS is well supported by the NumPy results, especially on aggregation operations such as centroid and extent. Burn with its parallel lazy execution, with autotuning and kernel fusion performed comparably well, especially on the CPU for distance and affine.

6. Discussion

In general, the results show that the SoA layout is equal to or faster than the AoS layout, especially when advanced vector extension instructions are available. Further autovectorization and vectorization-optimized libraries, such as Arrow, exhibit very strong performance, often surpassing the naive SIMD implementation. This shows that the used compiler infrastructure is very capable, and changing the data model and program structure to enable its optimizations is preferred before venturing into custom SIMD implementations, as is prominently demonstrated by the NumPy aggregation result.

Similarly, the results from Burn on the GPU and the CPU were reasonably fast. A key factor impacting these results is the integrated autotuning and kernel fusion. The latter is expected to become even more meaningful as kernels become more complex, consisting of multiple basic operations. Additionally, the CPU backend based on MLIR runs kernels in parallel across all available cores, using a single implementation that is comparable in code and syntax to NumPy. Even though MLIR and Burn are very young projects, and the CPU backend is still in an experimental state with room for improvement, the results already indicate it is a very interesting and promising approach.

7. Conclusion

In this work, we investigated array programming and vectorization for spatial operations. The evaluation shows that a columnar coordinate layout can substantially speed up certain spatial operations without any regressions compared to an interleaved layout. By relying on a modern systems programming language and compiler infrastructure, we further demonstrated that these performance improvements are accessible without writing specialized code. The comparison with widely used spatial processing libraries highlights the untapped potential for more efficient, sustainable spatial computing. Even though the experimental evaluation is rather epidermic, it nevertheless provides an indication of performance realities that are readily available and anticipated to become even more widely available in the future.

Data and Software Availability

The code used to get the results in Chapter 5 can be made available upon request.

Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 507196470.

References

Anderson, A., Malik, A., Gregg, D., 2016. Automatic Vectorization of Interleaved Data Revisited. *ACM Transactions on Architecture and Code Optimization*, 12(4), 1–25. https://dl.acm.org/doi/10.1145/2838735.

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., Schaub, T., 2016. The GeoJSON Format. Technical Report RFC7946, RFC Editor.

Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D., 2012. Neural acceleration for general-purpose approximate programs. *2012 45th annual IEEE/ACM international symposium on microarchitecture*, IEEE, 449–460.

Flynn, M., 1966. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), 1901–1909. http://ieeexplore.ieee.org/document/1447203/.

Habich, D., Pietrzyk, J., 2024. SIMDified Data Processing - Foundations, Abstraction, and Advanced Techniques. *Companion of the 2024 International Conference on Management of Data*, ACM, Santiago AA Chile, 613–621.

Kristof, P., Yu, H., Li, Z., Tian, X., 2012. Performance Study of SIMD Programming Models on Intel Multicore Processors. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2423–2432.

Kurzweil, R., 2001. The law of accelerating returns. *Alan Turing: Life and legacy of a great thinker*, Springer, 381–416.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O., 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, Seoul, Korea (South), 2–14.

OGC/ISO, 2011. Simple Feature Access – Part 1: Common Architecture.

Polychroniou, O., Raghavan, A., Ross, K. A., 2015. Rethinking SIMD Vectorization for In-Memory Databases. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, Melbourne Victoria Australia, 1493–1508.

Rayhan, Y., Aref, W. G., 2023. SIMD-ified R-tree Query Processing and Optimization. *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*, ACM, Hamburg Germany.

Saeedan, M., Eldawy, A., 2022. Spatial Parquet: A Column File Format for Geospatial Data Lakes. *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*, ACM, Seattle Washington, 1–4.