Structured vs. Unstructured Grids: Evaluating the Impact of Lookup Tables and Formulas on Performance

High Performance Computing for Weather and Climate - Work Project

Eva Glock, Malin Locher, Leonard Mantel

August 25, 2025

1 Introduction and Objectives

Weather and climate models compute atmospheric variables by solving the Euler equations on computational grids (Prein, 2025). By this, such models provide weather forecasts and climate projections, being a crucial service to the economy and society (Williamson et al., 2002).

The computational grid on which the atmospheric equations are solved can be constructed in diverse ways. A common grid type is the structured regular latitude/longitude grid (Staniforth & Thuburn, 2012). This regular array is ordered such that accessing the individual grid points is intuitive according to an (i, j, k) ordering (Thompson & Weatherill, 1992). Yet, a disadvantage of such grids is the so-called Pole Problem: Near the poles, the longitudinal grid spacing approaches zero – inducing not only stability issues but also data communication bottlenecks as the close location of many grid points near the poles requires enhanced communication between processors (Staniforth & Thuburn, 2012).

To overcome these challenges, many models are implemented using unstructured grids. Unlike structured grids, unstructured grids do not follow a predefined (i, j, k) ordering, meaning the order of their cells must be explicitly determined (Thompson & Weatherill, 1992). Various types of unstructured grids have been developed. One example is the geodesic icosahedral grid, which is composed of numerous triangles that allow for local refinement and is used in the ICON model (Prein, 2025).

Since accessing grid points in unstructured grids is not straightforward, specialized methods are required to efficiently handle this task. One possible approach to arrange an unstructured, multidimensional grid in an accessible way is the use of space-filling curves (SFCs), continuous curves that traverse every cell in the array (Aftosmis et al., 2004; Sergeyev et al., 2013). These curves map multidimensional arrays into a one-dimensional space (Aftosmis et al., 2004). Typically, SFCs work by recursively subdividing a field into smaller subfields, with a predefined curve pattern connecting these subfields (Gutermuth, 2014). Each of these subfields can be further divided, enabling grid refinement where needed (Gutermuth, 2014). Two commonly employed SFCs following this approach are Z-order and the Hilbert curves (Gutermuth, 2014). Using SFCs to access unstructured grids is beneficial because SFCs yield high locality of the grid (Aftosmis et al., 2004). Moreover, accessing grid points requires only local information rather than knowledge of the entire SFC mapping of the grid (Aftosmis et al., 2004). Therefore, the set-up of SFCs is computationally not expensive and their employment results in efficient memory use and low run times (Aftosmis et al., 2004; Gutermuth, 2014).

The Z-order curve is a SFC that uses the letter “Z” as its base pattern (see Figure 1) (Bader, 2013). It provides high locality, however, not all neighbors in the Z-order curves are also neighboring grid cells (Gutermuth, 2014). Furthermore, the curve follows a symmetric pattern, making the computation of the grid straightforward (Gutermuth, 2014).

The Hilbert curve, in contrast, follows the pattern of a “U” (Figure 2) (Aftosmis et al., 2004). The main advantage of this curve is its enhanced locality since, unlike for the Z-order curve, all neighbors in the Hilbert curve are also neighbors on the grid (Gutermuth, 2014). Yet, the curve pattern is not entirely symmetric but also includes rotations, making the computation of the curve somewhat more complex (Gutermuth, 2014).

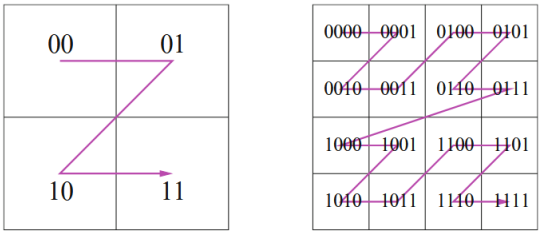


Figure 1: The Z-order curve on a 2×2 and a 4×4 grid.
From Bader (2013).

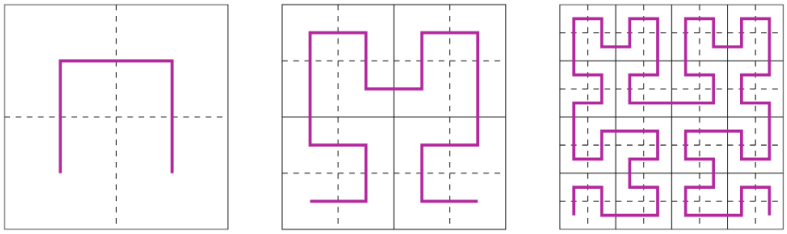


Figure 2: The Hilbert curve on a 2×2, a 4×4, and a 16×16 grid.
From Bader (2013).

Implementing SFCs as accessing methods for unstructured grids can be done in two ways: by explicitly computing the required index for each access using a formula, or indirectly via a lookup table.

Different structured and unstructured grids vary in their efficiency of memory access, performance, and runtime. Since weather and climate models process vast amounts of data on large grids, efficient management of computational resources is crucial (Behrens, 2005; Gutermuth, 2014) Runtime is particularly critical for weather forecasts, as projections must be available before the actual weather events occur (Behrens, 2005).

Therefore, this project aims to implement different versions of unstructured grids. We analyze them following the question of how they compare to a structured latitude/longitude grid in terms of memory accesses, performance, and runtime. The unstructured grids implemented include formula and lookup table versions of the Z-order curve and a j-stride SFC that follows the pattern of the regular grid, as well as a lookup table version of a randomly ordered grid. These unstructured grids along with an implementation of a lookup table version of the Hilbert curve by Wong et al. (2024) will be compared across different grid sizes with respect to runtime, arithmetic intensity, performance, and bandwidth. In a final step, we will establish whether these different grid versions are memory- or compute-bound based on the Roofline model by Williams et al. (2009).

2 Methods

2.1 Structured grid implementation

As a foundation for our work, we used the `stencil2d` Fortran code developed during the lecture sessions. Since our project focuses on exploring various unstructured grid implementations, we chose a simple stencil version that includes inlining but does not include additional performance

optimizations such as OpenMP or CPU-specific enhancements. The structured grid reference model against which all other versions are compared is the `stencil2d-structured.F90` file, which is a copy of the `stencil2d-inlining.F90` file from day 1 of the block course.

2.2 Unstructured grid implementations

We developed and analyzed five different implementations of unstructured grids, based on three distinct grid point ordering strategies. Where feasible, each ordering was implemented using both a lookup table and a direct formula to access neighboring grid points, allowing us to compare their performance. The implemented Fortran versions are as follows:

1. `stencil2d-jstride_lookup.F90`:
   Sequential j-stride grid ordering with neighbor access via a lookup table.
   J-stride ordering maps the grid points in a row-major layout, following the natural indexing in the j (row) direction. This ordering preserves spatial locality and serves as a baseline among unstructured approaches.

2. `stencil2d-jstride_formula.F90`:
   Sequential j-stride grid ordering with neighbor access via an explicit formula.

3. `stencil2d-zcurve_lookup.F90`:
   Z-order curve grid ordering with neighbor access via a lookup table.
   Z-curve ordering maps multi-dimensional data to one dimension while preserving spatial locality to a greater extent than random or row-major orderings.

4. `stencil2d-zcurve_formula.F90`:
   Z-order curve grid ordering with neighbor access via an explicit formula.

5. `stencil2d-random_lookup.F90`:
   Random grid ordering with neighbor access via a lookup table.
   In this approach, the grid points are randomly reordered. This method serves to evaluate performance impacts in the absence of spatial locality.

Additionally, we included the `stencil2d-hilbert.F90` version from last year's project by Wong et al. (2024) in our analysis. However, we neither modified this code nor contributed to its development.
For performance reasons, we used separate Fortran files for each version. The single-file approach performed worse because it required numerous if-statements, which introduced additional overhead. However, execution of all versions is handled through a single Python script (`launcher.ipynb`).

To convert 2D grid coordinates (i, j) into a 1D index n, different mapping strategies are employed. The transformation from structured to unstructured indexing is defined by the following formulas:

- j-stride:

$$n = i + (j - 1) \cdot nx \tag{1}$$

- Z-order curve:

$$n = \text{interleave\_bits}(i, j) \tag{2}$$

- Random:

$$n = \text{random}(i, j) \tag{3}$$

The primary distinction between the lookup and formula versions lies in how neighboring grid points are accessed during the stencil computation. In the lookup versions, a neighbor table is generated and initialized once before execution. This table stores the indices of the four direct neighbors (top, bottom, left, right) for each grid point. During the stencil update, neighbor indices are retrieved directly from the table. In the formula versions, neighbor indices are computed on the fly during each access, using the formula specific to the respective ordering strategy.

It is important to note that in all of our unstructured grid implementations, the halo update was replaced by periodic boundary conditions implemented using the modulo operator. This approach is consistent with the Hilbert curve version developed by last year's group.

2.3 Validation

To validate the correctness of each unstructured grid implementation, we compare its output to that of the baseline structured version. Specifically, we compute the difference between the result of each version and the baseline, and evaluate the mean squared error (MSE) as a quantitative measure of deviation.
In addition to the numerical evaluation, we also provide visualizations of the differences to facilitate qualitative assessment and to help identify potential structural errors or anomalies in the output.

2.4 Performance Analysis

For the performance analysis, several hardware and computational metrics were collected using the `perf_wrap.sh` shell script provided during the course. This script was extended to also measure floating-point operations (flop) by adding the necessary performance counters.
The following primary metrics were recorded:

- Execution time (s)
- Total memory access (GB)
- DRAM memory access (GB)
- Total number of floating-point operations (flop)

From these primary measurements, additional derived performance metrics were computed:

$$\text{Memory Bandwidth (GB/s)} = \frac{\text{Total memory access}}{\text{Execution time}} \tag{4}$$

$$\text{Arithmetic intensity (flop/byte)} = \frac{\text{Total number of flop}}{\text{Total memory access}} \tag{5}$$

$$\text{Computational performance (Gflop/s)} = \frac{\text{Total number of flop}}{\text{Execution time}} \tag{6}$$

These values serve as the basis for constructing the roofline model, which is used to analyze the performance characteristics of each implementation in relation to hardware limitations. The roofline model provides insight into whether an implementation is memory-bound or compute-bound by visualizing a version's computational performance and arithmetic intensity along with ceilings for the DRAM and cache memory accesses and the computations. This helps identifying performance bottlenecks.

All versions were run on different square grid sizes, with (nx, ny) = (32, 32), (64, 64), (128, 128), (256, 256) and (512, 512), using $nz = 64$ over 1024 iterations. Performance on non-square grids was not considered.
The analysis was performed in two steps. First, a single execution was carried out for each version and grid size. In the second step, $r = 10$ executions were performed for all grid sizes to assess variability. For (nx, ny) = (512, 512) the r-times executions were skipped due to time constraints.

3 Results

3.1 Validation

The final 3D Cartesian grids of all versions implemented by us are equivalent to the 3D Cartesian grid outputted by the structured version, when removing the halo from the structured grid. The MSE between each unstructured version and the structured version is 0. An exception is the grid generated by the Hilbert implementation, which shows a slight deviation from the structured grid. For a grid size of (nx, ny) = (128, 128), $nz = 64$, and 1024 iterations, the MSE amounts to $4.448 \cdot 10^{-15}$. This deviation is considered negligible and can be ignored for practical purposes.

3.2 Performance

A summary of the results from a single execution across all grid sizes is provided in Table 1 in the appendix. Below, we present an overview of the results for the grid size (nx, ny) = (128, 128). These findings are representative for other grid sizes, as similar trends were observed for them. Furthermore, results from $r = 10$ repeated executions (Figure 6 in Appendix) indicate that a single execution reliably reflects overall performance, as the variations between runs are minimal. Since our analysis focuses on the overall trends in performance and not the exact values, the following results present values from a single execution rather than an average.

As can be seen on Figure 3, the total memory access, DRAM memory access, and execution time exhibit to some extent similar patterns. Across these three parameters, very low values were measured for the structured grid, with a total memory access of 15 GB, a DRAM memory access of 0.04 GB, and an execution time of 0.64 s. Our own lookup table implementations of the unstructured grids (j-stride, Z-order curve, and random) perform second best, with the same DRAM access as for the structured grid and total memory accesses and execution time amounting to three to four times the structured grid value. For the Hilbert curve lookup table version by Wong et al. (2024), results are mixed across parameters. While the DRAM access remains as low as in

the other lookup versions and the execution time is only slightly higher, the total memory access is significantly larger, reaching up to 90 GB. Both formula versions have higher values than the structured grid and our own lookup versions across the three parameters. The j-stride formula shows a total memory access of 56 GB, which is slightly higher than our lookup implementations but still considerably lower than the Hilbert version. Its DRAM memory access is twice as high, and its execution time is ten times longer compared to our lookup versions. Overall, the Z-order curve formula performs worst, with a total memory access of 112 GB, a DRAM memory access of 0.3 GB, and an execution time exceeding two minutes. The memory bandwidth is lowest for the formula implementations. For the Z-order curve, the memory bandwidth is particularly low at 0.82 GB/s, while the j-stride curve reaches 2.74 GB/s. For the implementations of the structured grid as well as our own look-up table unstructured grid implementations, the memory bandwidths are relatively similar, ranging from 21 GB/s to 25 GB/s. The Hilbert curve has the highest memory bandwidth with more than 34 GB/s.

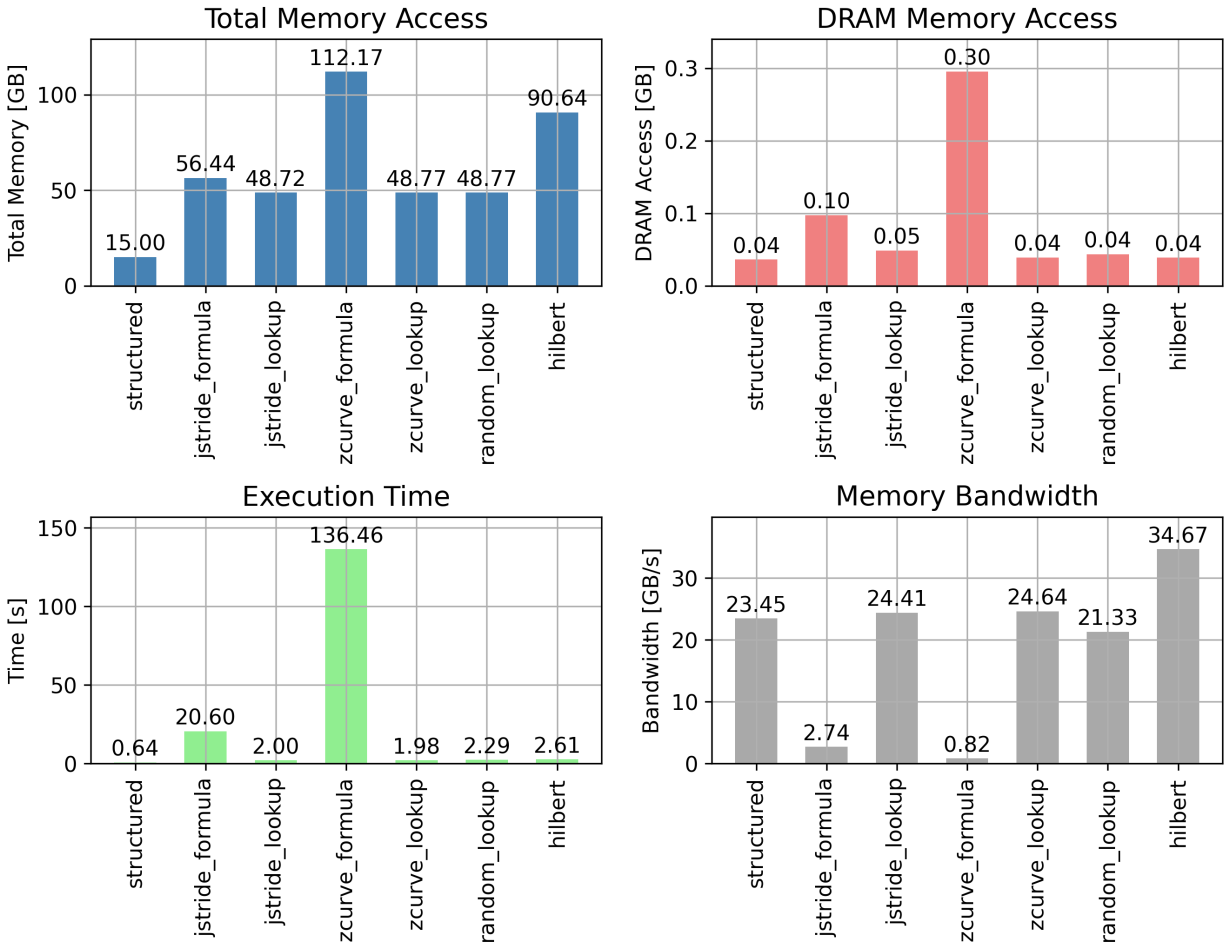


Figure 3: Comparison of memory-related metrics for $(nx, ny) = (128, 128)$

Our analysis showed that the measured number of floating-point operations (flops) is not consistent across all versions. Theoretically, this value should be identical for all versions, as both the lookup and formula implementations perform the same number of floating-point operations. The formula versions (and the Hilbert version, due to differences in implementation) involve a higher number of integer operations, which may contribute to discrepancies. A likely cause is that the

function used within the `perf_wrap.sh` script does not reliably distinguish between integer and floating-point operations.
To ensure a fair comparison of arithmetic intensity and computational performance, we corrected the flop count for versions with abnormally high values by setting them to the mean of the other versions, which were very similar. In Figure 4, the corrected values are shown as solid bar segments, while the hatched areas above them represent the actual measured values.

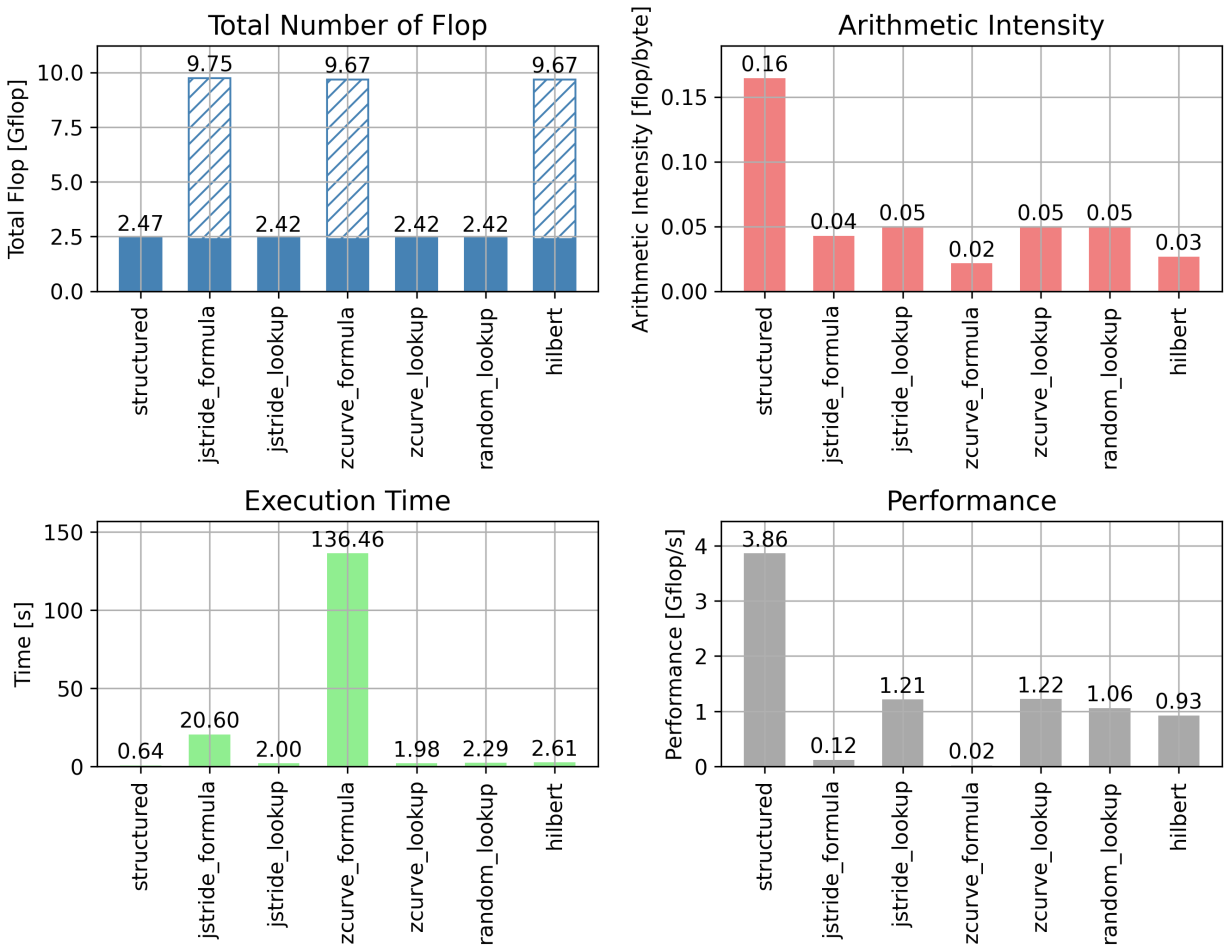


Figure 4: Arithmetic and performance characteristics for $(nx, ny) = (128, 128)$
Note: Execution time shown again because relevant in the computation of performance.

Arithmetic intensity and computational performance exhibit similar patterns across the different grid versions. The structured version achieves the highest values, with an arithmetic intensity of 0.16 flop/byte and a computational performance of 3.86 Gflop/s.
The lookup table versions implemented by us show comparable results, each with an arithmetic intensity of 0.05 flop/byte and computational performance ranging from 1.06 to 1.22 Gflop/s. The Hilbert curve implementation performs slightly worse, with an arithmetic intensity of 0.03 flop/byte and a computational performance of 0.93 Gflop/s. The j-stride formula version has a similar arithmetic intensity of 0.04 flop/byte but a much lower computational performance of only 0.12 Gflop/s. The Z-order curve formula implementation performs worst with an arithmetic intensity of 0.02 flop/byte and a computational performance of only 0.02 Gflop/s.

As can be seen on the Roofline Model (Figure 5), all grid implementations are memory-bound.

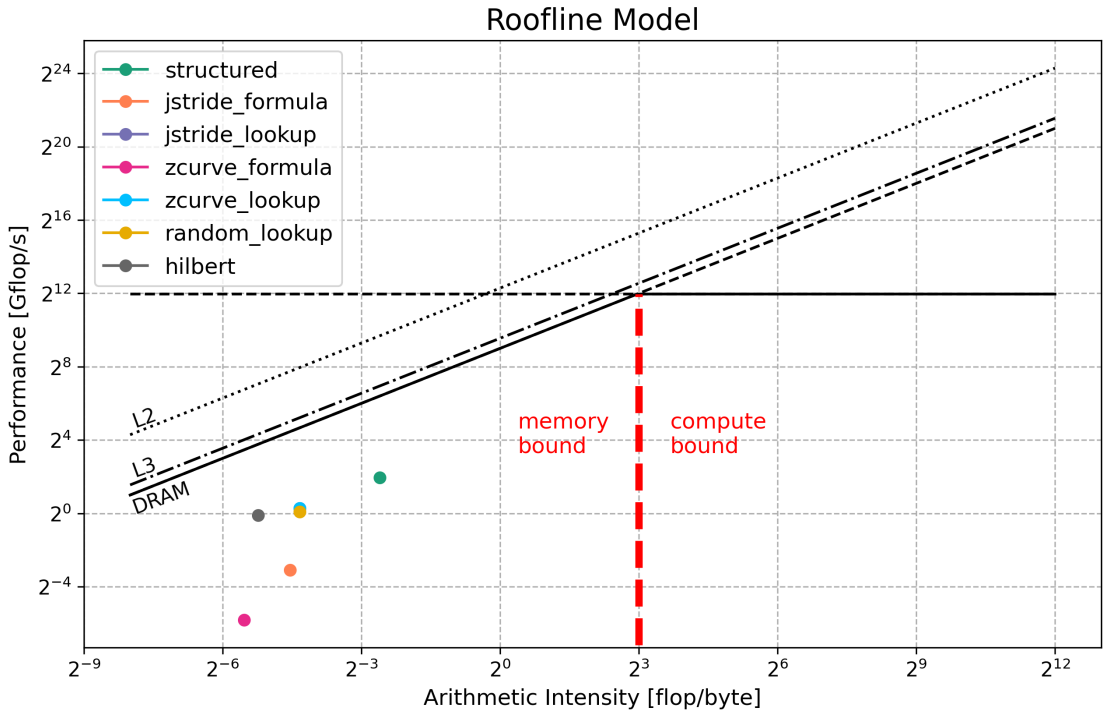


Figure 5: Roofline Model for $(nx, ny) = (128, 128)$
Note: Data points of the j-stride, Z-order curve, and random lookup versions are very close to one another.

The formula implementations are the furthest away from the ceiling, with the Z-order curve performing worst for both, arithmetic intensity and performance. The lookup table versions and the structured grid seem to be located on a line parallel to the ceiling, with the Hilbert curve being furthest from and the structured grid being closest to the memory-compute bound boundary.

3.3 Other grid sizes

As mentioned above, the overall patterns remain the same across different grid sizes. However, a few observations should be noted:
When the grid size is doubled (e.g., from (nx, ny) = (64, 64) to (nx, ny) = (128, 128)):

- Execution time, total memory access, and the total number of floating-point operations (flops) increase by approximately a factor of four.
- DRAM memory access rises sharply for very large grid sizes but remains close to zero for small grids.
- Memory bandwidth remains approximately constant, although measured values show a slight decrease as grid size increases.
- Arithmetic intensity remains approximately constant.
- Computational performance also remains approximately constant, although measured values show a slight decrease as grid size increases.

The comparison of the different grid sizes is shown in Table 1, and Figures 7 - 10 in the Appendix.

4 Discussion and Conclusion

4.1 Structured versus unstructured grids

Our observations show that the structured grid outperforms our best-performing lookup version, the unstructured j-stride lookup implementation, across all measured parameters. Based on this, we could conclude that the structured grid generally performs better than an unstructured grid. However, it is important to note that this project only considered simple, quadratic grids. For more complex grid configurations, structured grids may no longer be feasible due to the lack of straightforward indexing. Moreover, unstructured grids offer distinct advantages over structured grids, such as avoiding the pole problem and reducing communication bottlenecks, as discussed in the introduction.

4.2 Comparison of unstructured grids

When comparing our implementations of unstructured grid lookup versions (j-stride, Z-order curve and random lookup), we observe very similar performance across most parameters.
Execution times are nearly identical, with the random version being slightly slower than the other two. The total number of memory accesses and DRAM accesses is also similar across the three versions for grid sizes up to (nx, ny) = (128, 128). The j-stride lookup version shows slightly lower total memory access, likely because its lookup table construction is less memory-intensive. For larger grids, such as (nx, ny) = (256, 256), the DRAM accesses for the random version increase significantly. This is expected, as both the j-stride and Z-order curve store data with an underlying logic that can lead to more favorable memory access patterns. The sharp increase in DRAM accesses for large grids suggests that the entire grid no longer fits into the cache, resulting in more cache misses and expensive DRAM accesses, which negatively impact performance. For a (512, 512) grid, even less of the grid fits into the cache, again much increasing the DRAM accesses for the random grid. Furthermore, for such a large grid, the DRAM memory accesses also increase considerably for the other unstructured grids, likely because some of the neighbors need to be loaded from the DRAM. Interestingly, the Z-order curve formula version has less cache misses than the jstride formula version, indicating the enhanced spatial locality of the Z-order curve.
The arithmetic intensity remains the same across all three versions, as the total number of flops and total memory access are approximately equal. Similarly, memory bandwidth and computational performance are almost identical but slightly lower for the random version due to its higher execution time.

The Hilbert curve lookup table, which was not implemented by our group, performs worse than our implementations. It exhibits higher execution time, greater total memory access, and lower arithmetic intensity and computational performance, even when compared to the random lookup version. This is a result that was not expected as, due to its enhanced data locality, the Hilbert curve should perform better than the random curve implementation (Gutermuth, 2014). These differences are likely due to implementation issues. Upon reviewing the code, we suspect that last year's group may have unintentionally combined a lookup table approach along with a formula-based implementation, rather than exclusively using a lookup table. This assumption is supported by the observation that the Hilbert version has the same number of measured flops as our formula

versions. Interestingly, the Hilbert version also reports the highest memory bandwidth. However, this is primarily a consequence of its very high total memory access and does not translate into a performance advantage.

Examining our formula-based implementations, we observe that the j-stride formula version consistently outperforms the Z-order curve formula version across all parameters. This outcome is expected, as the Z-order curve is computationally more expensive. Calculating the Z-order curve involves repeated modulo operations, which are known to be both time- and memory-intensive (Verloop et al., 2023). Consequently, memory bandwidth, arithmetic intensity, and computational performance are all lower for the Z-order curve due to its significantly higher execution time.

Gutermuth (2014) also found that for a cubical grid, which somewhat resembles the regular latitude/longitude grid, SFCs did not provide an advantage compared to the initial ordering method used, which aligns with our findings. However, their study also demonstrated that for a hexagonal grid, both the Hilbert and Z-Order SFCs provide clear performance advantages over the original grid accessing method.

4.3 Lookup table versus formula implementations

When comparing the formula implementations to the lookup table versions, it is evident that the lookup approaches are considerably more advantageous. This is because formula-based methods require computations for every grid cell and each of its neighbors, whereas the lookup tables store precomputed neighbor information. Furthermore, the data in lookup tables can be efficiently reused and can benefit from more cache locality, resulting in improved performance.

4.4 Outlook and conclusion

The Roofline model shows that all grid implementations are memory-bound. In line with the previous argumentation, the Roofline model further demonstrates that the structured grid uses resources most efficiently while the Z-order curve formula implementation is the most expensive. Interestingly, the lookup versions and the structured grid seem to be located on a line parallel to the memory-bound ceiling, potentially indicating an additional performance-limiting ceiling. Future research can focus on determining the root of this limitation in performance. Until more information on an additional ceiling is available, and because all grid versions are memory-bound, we recommend that future grid improvements should focus on enhancing the efficient usage of memory and thus decreasing cache misses.

In conclusion, for this project using a quadratic 2D grid setup, the structured grid clearly outperformed all of our unstructured grid implementations. However, as noted earlier, for more complex grid configurations, structured grids may no longer be feasible due to their lack of straightforward indexing. We also conclude that, when employing an unstructured grid, implementations using lookup tables achieve better performance than formula-based approaches, where neighbors are recalculated repeatedly.

Bibliography

Aftosmis, M., Berger, M., & Murman, S. (2004). Applications of space-filling-curves to cartesian methods for CFD. In *42nd AIAA aerospace sciences meeting and exhibit*. American Institute of Aeronautics; Astronautics. https://doi.org/10.2514/6.2004-1232

Bader, M. (2013). *Space-filling curves* (9th ed.). Springer. https://doi.org/10.1007/978-3-642-31046-1

Behrens, J. (2005). Multilevel optimization by space-filling curves in adaptive atmospheric modeling. *18th ASIM Symposium on Simulation Techniques*. https://epic.awi.de/id/eprint/15120/1/Beh2005b.pdf

Gutermuth. (2014). *Application of space-filling curves on unstructured grids for simulations on a sphere* [Bachelor's thesis]. Technische Universität München. https://www.martin-schreiber.info/data/student_projects/2014_BA_dominik_gutermuth.pdf

Prein, A. (2025, March 20). *Horizontal discretizations in atmospheric models* [Lecture slides]. https://polybox.ethz.ch/index.php/s/xur4x0x7pWUSr1R

Sergeyev, Y. D., Strongin, R. G., & Lera, D. (2013). *Introduction to global optimization exploiting space-filling curves*. Springer. https://doi.org/10.1007/978-1-4614-8042-6

Staniforth, A., & Thuburn, J. (2012). Horizontal grids for global weather and climate prediction models: A review. *Quarterly Journal of the Royal Meteorological Society*, *138*(662), 1–26. https://doi.org/10.1002/qj.958

Thompson, J. F., & Weatherill, N. P. (1992). Structured and unstructured grid generation. In T. C. Pilkington, B. Loftis, T. Palmer, & T. F. Budinger (Eds.), *High-performance computing in biomedical research* (pp. 63–111). CRC Press. https://doi.org/https://doi.org/10.1201/9781003068136

Verloop, M., Koopman, T., & Scholz, S.-B. (2023). Modulo in high-performance code: Strength reduction for modulo-based array indexing in loops. *The 35th Symposium on Implementation and Application of Functional Languages*, 1–13. https://doi.org/10.1145/3652561.3652573

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65–76. https://doi.org/10.1145/1498765.1498785

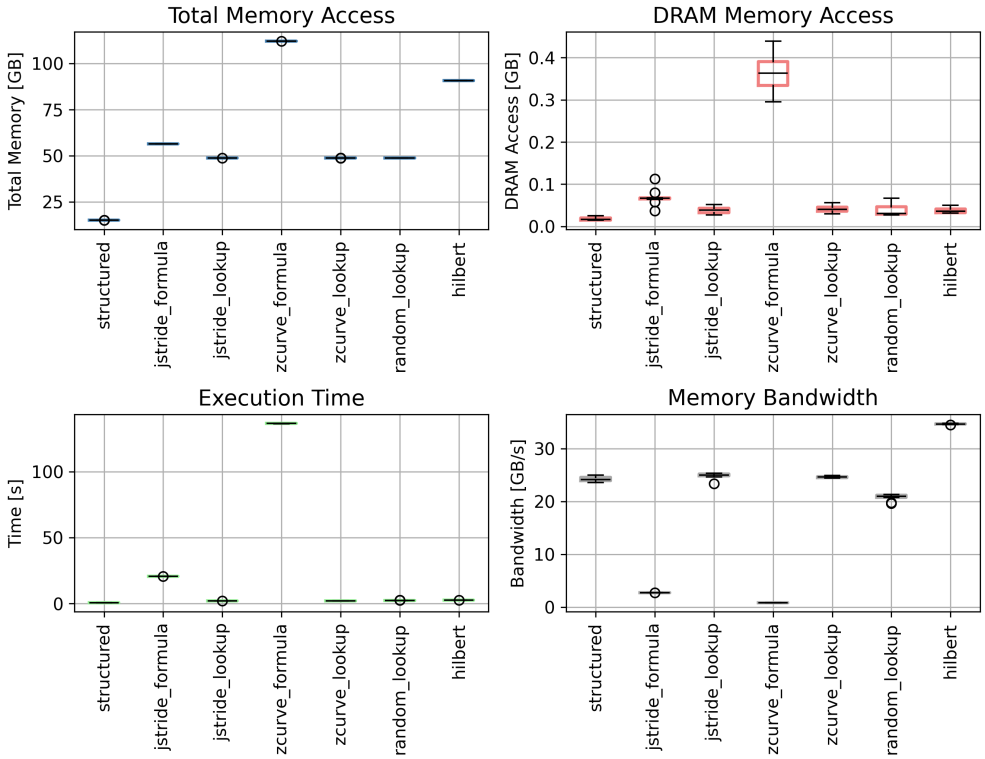
Williamson, R. A., Hertzfeld, H. R., & Cordes, J. (2002). *The socio-economic value of improved weather and climate information*. Space Policy Institue, George Washington University. Washington DC. https://repository.library.noaa.gov/view/noaa/70630/noaa_70630_DS1.pdf

Wong, T. C., Ghielmini, C., & Nakamura, M. (2024). *stencil2d-hilbert.F90* [Fortran Code from HPC4WC 2024 ETH class]. https://github.com/ofuhrer/HPC4WC/blob/main/projects/2024/project10_structured_vs_unstructured2/stencil2d-hilbert.F90

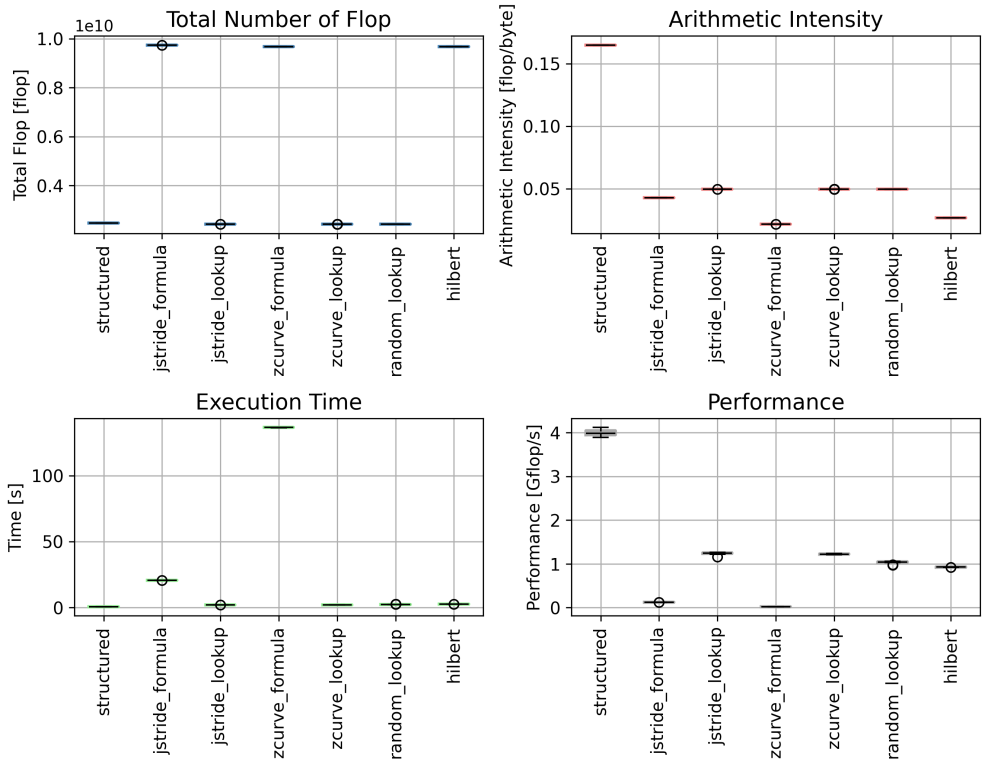
AI statement

The large language models Google Copilot and ChatGPT (GPT-4) were used to support code troubleshooting and debugging during the development process. Furthermore, in preparing this report, we used ChatGPT (GPT-4 and GPT-5) and Perplexity (based on GPT-4) to assist with language refinement, grammar correction, and structural improvements. In addition, Chat GPT (GPT-4 and GPT-5) was also used to support the layouting of this report and for correcting some entries in the reference list. All original content, ideas, and substantive writing are our own, AI was not used to generate new content.

Appendix



(a) Memory related metrics



(b) Arithmetic and performance characteristics

Figure 6: Boxplots of memory related metrics and arithmetic and performance characteristics fo $r = 10$ executions for $(nx, ny) = (128, 128)$

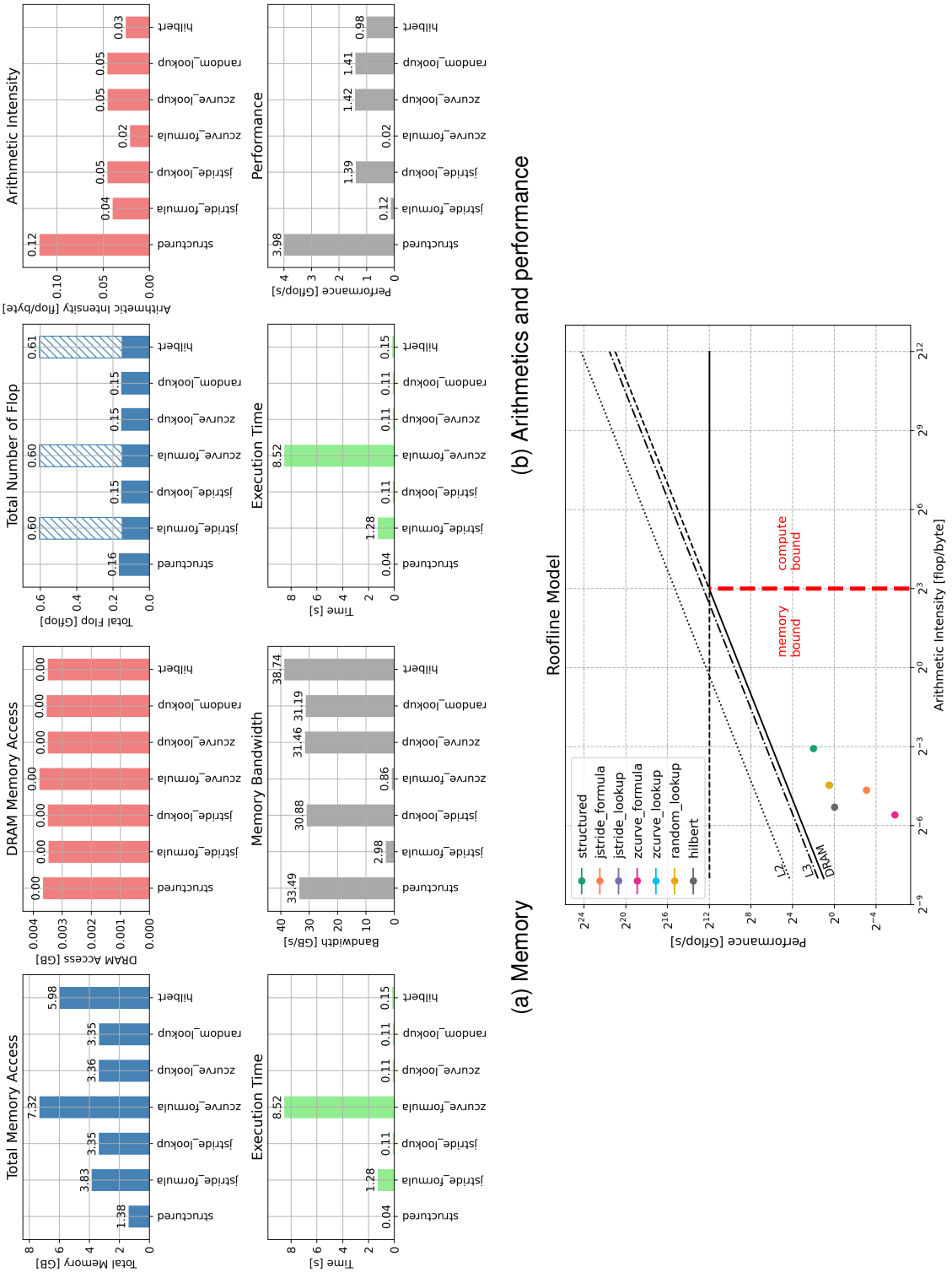


Figure 7: Memory-related metrics, arithmetic, and performance characteristics and roofline model for $(nx, ny) = (32, 32)$

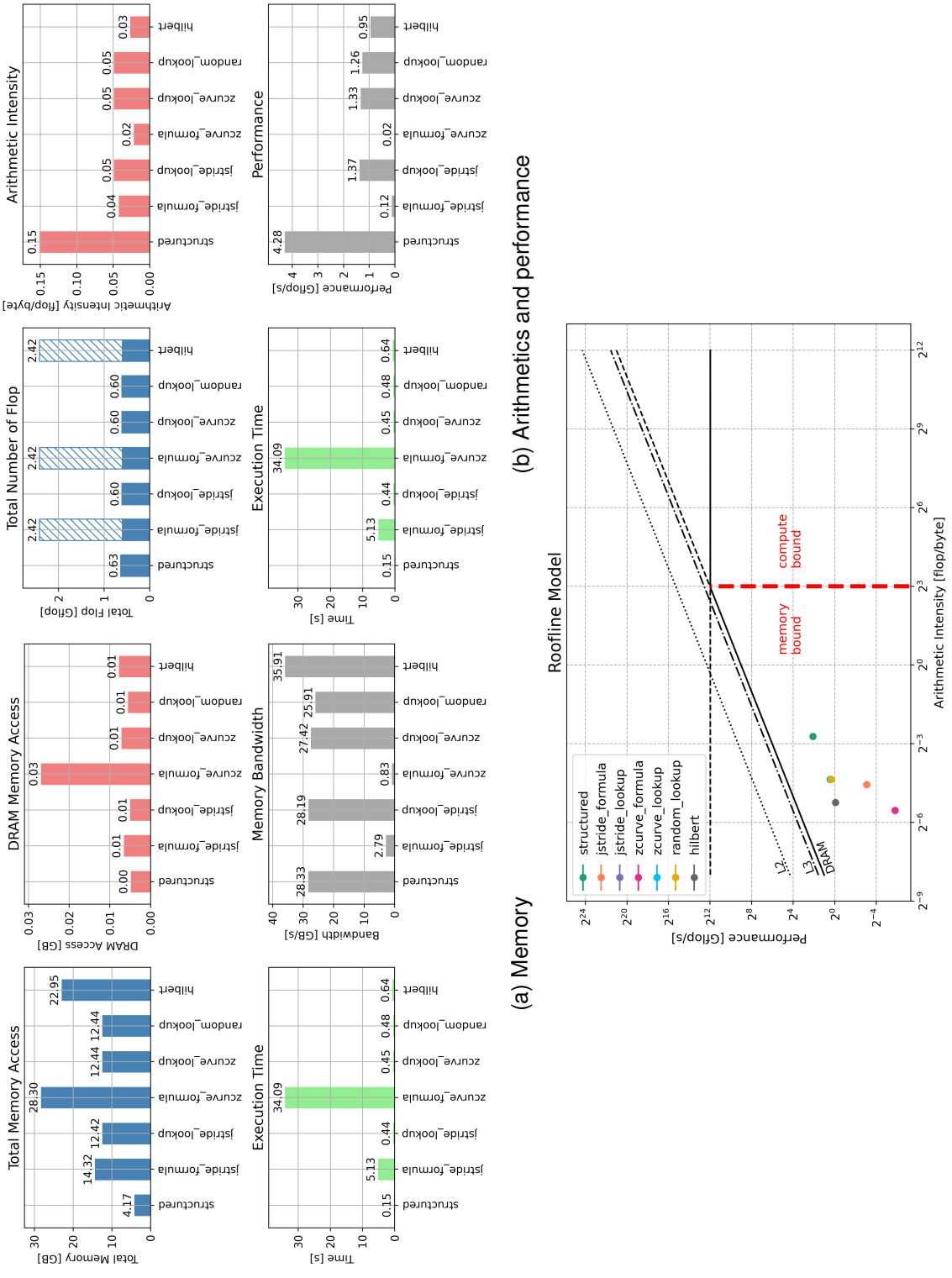


Figure 8: Memory-related metrics, arithmetic, and performance characteristics and roofline model for $(nx, ny) = (64, 64)$

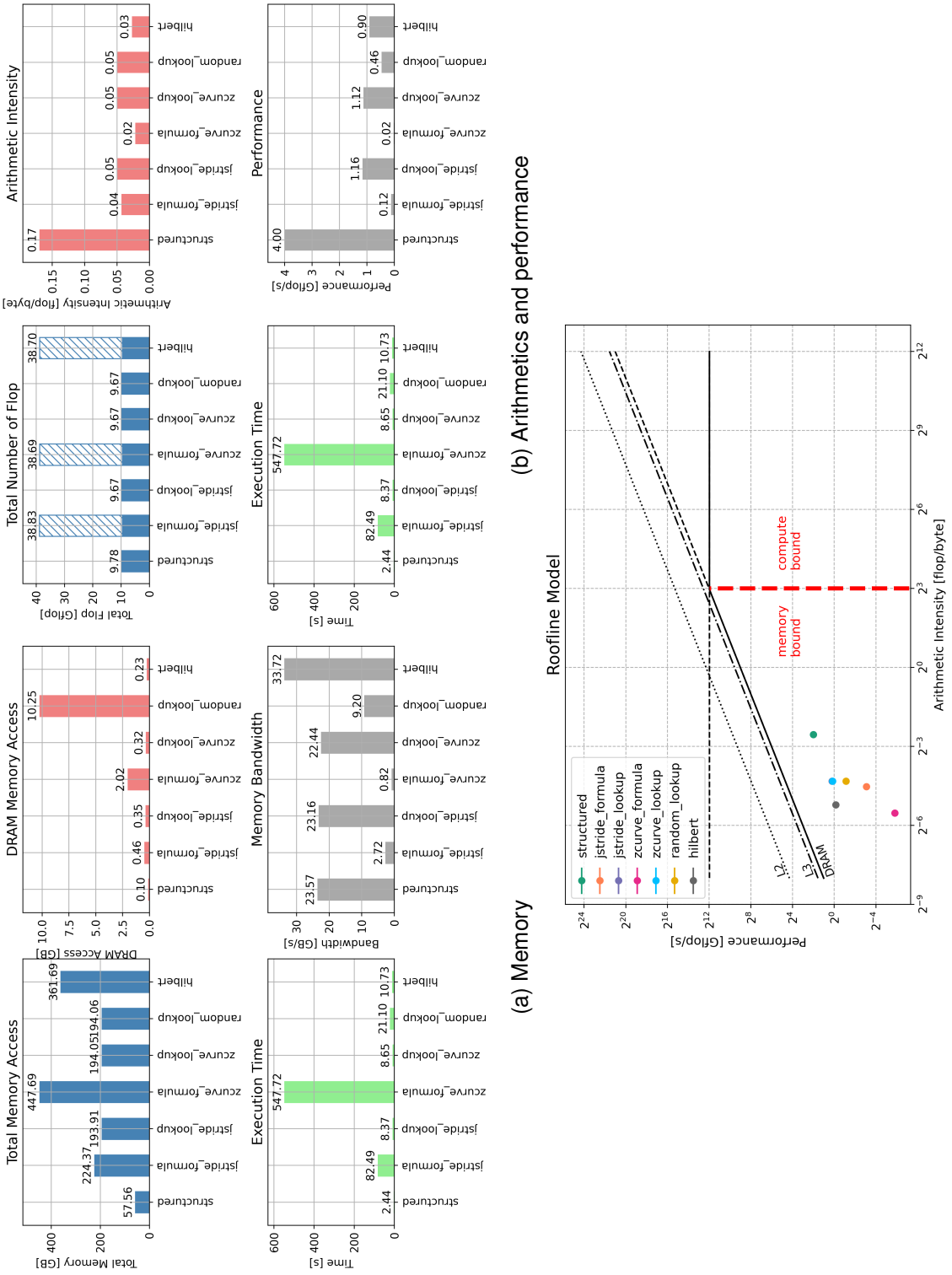


Figure 9: Memory-related metrics, arithmetic, and performance characteristics and roofline model for $(nx, ny) = (256, 256)$

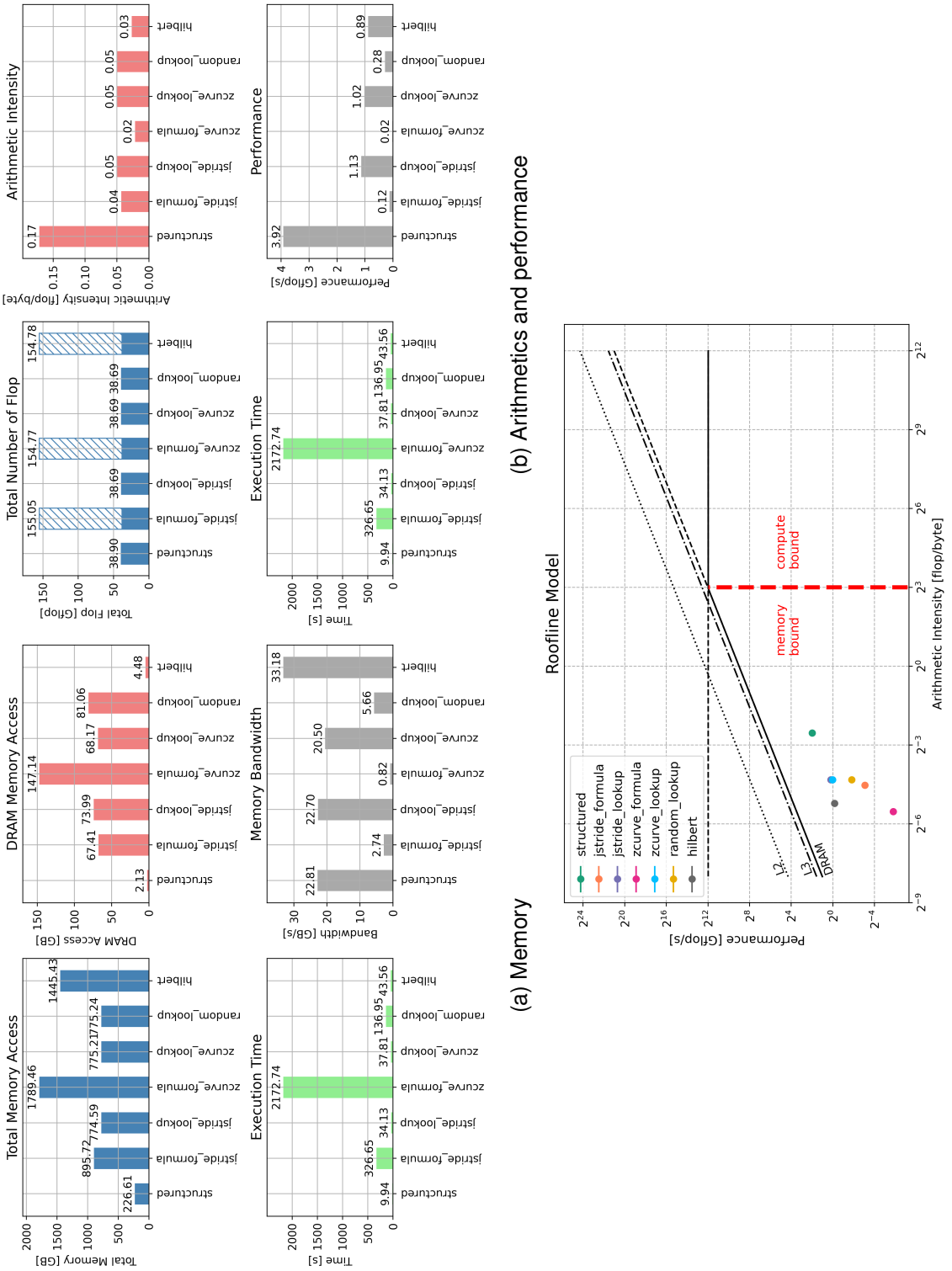


Figure 10: Memory-related metrics, arithmetic, and performance characteristics and roofline model for $(nx, ny) = (512, 512)$

nx, ny	Variable	structured	jstride_formula	jstride_lookup	zcurve_formula	zcurve_lookup	random_lookup	hilbert
32, 32	Execution Time (s)	0.04	1.28	0.11	8.52	0.11	0.11	0.15
	Total Memory Access (GB)	1.38	3.83	3.35	7.32	3.36	3.35	5.98
	DRAM Memory Access (GB)	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Memory Bandwidth (GB/s)	33.49	2.98	30.88	0.86	31.46	31.19	38.74
	Total Number of Flop (flop)	0.16	0.60 (0.15)	0.15	0.60 (0.15)	0.15	0.15	0.61 (0.15)
	Arithmetic Intensity (flop/byte)	0.12	0.04	0.05	0.02	0.05	0.05	0.03
	Performance (Gflop/s)	3.98	0.12	1.39	0.02	1.42	1.41	0.98
64, 64	Execution Time (s)	0.15	5.13	0.44	34.09	0.45	0.48	0.64
	Total Memory Access (GB)	4.17	14.32	12.42	28.30	12.44	12.44	22.95
	DRAM Memory Access (GB)	0.00	0.01	0.01	0.03	0.01	0.01	0.01
	Memory Bandwidth (GB/s)	28.33	2.79	28.19	0.83	27.42	25.91	35.91
	Total Number of Flop (flop)	0.63	2.42 (0.60)	0.60	2.42 (0.60)	0.60	0.60	2.42 (0.60)
	Arithmetic Intensity (flop/byte)	0.15	0.04	0.05	0.02	0.05	0.05	0.03
	Performance (Gflop/s)	4.28	0.12	1.37	0.02	1.33	1.26	0.95
128, 128	Execution Time (s)	0.64	20.60	2.00	136.46	1.98	2.29	2.61
	Total Memory Access (GB)	15.00	56.44	48.72	112.17	48.77	48.77	90.64
	DRAM Memory Access (GB)	0.04	0.10	0.05	0.30	0.04	0.04	0.04
	Memory Bandwidth (GB/s)	23.45	2.74	24.41	0.82	24.64	21.33	34.67
	Total Number of Flop (flop)	2.47	9.75 (2.42)	2.42	9.67 (2.42)	2.42	2.42	9.67 (2.42)
	Arithmetic Intensity (flop/byte)	0.16	0.04	0.05	0.02	0.05	0.05	0.03
	Performance (Gflop/s)	3.86	0.12	1.21	0.02	1.22	1.06	0.93
256, 256	Execution Time (s)	2.44	82.49	8.37	547.72	8.65	21.10	10.73
	Total Memory Access (GB)	57.56	224.37	193.91	447.69	194.05	194.06	361.69
	DRAM Memory Access (GB)	0.10	0.46	0.35	2.02	0.32	10.25	0.23
	Memory Bandwidth (GB/s)	23.57	2.72	23.16	0.82	22.44	9.20	33.72
	Total Number of Flop (flop)	9.78	38.83 (9.67)	9.67	38.69 (9.67)	9.67	9.67	38.70 (9.67)
	Arithmetic Intensity (flop/byte)	0.17	0.04	0.05	0.02	0.05	0.05	0.03
	Performance (Gflop/s)	4.00	0.12	1.16	0.02	1.12	0.46	0.90
512, 512	Execution Time (s)	9.94	326.65	34.13	2172.74	37.81	136.95	43.56
	Total Memory Access (GB)	226.61	895.72	774.59	1789.46	775.21	775.24	1445.43
	DRAM Memory Access (GB)	2.13	67.41	73.99	147.14	68.17	81.06	4.48
	Memory Bandwidth (GB/s)	22.81	2.74	22.70	0.82	20.50	5.66	33.18
	Total Number of Flop (flop)	38.90	38.69 (155.05)	38.69	38.69 (154.77)	38.69	38.69	38.69 (154.78)
	Arithmetic Intensity (flop/byte)	0.17	0.04	0.05	0.02	0.05	0.05	0.03
	Performance (Gflop/s)	3.92	0.12	1.13	0.02	1.02	0.28	0.89

Table 1: Summary of single-execution results for different domain dimensions and versions. Flop numbers in parentheses indicate the corrected values used in the calculation of arithmetic intensity and performance.