The American Journal of Engineering and Technology
ISSN 2689-0984 Volume 08 - 2026

Comparative Analysis of MSVC and Clang/LLVM Compilation on Windows on Arm

Gleb Khmyznikov
Senior Software Engineer, Microsoft Bellevue, WA, US

Received: 26 Jan 2026 | Received Revised Version: 14 Feb 2026 | Accepted: 24 Mar 2026 | Published: 30 Apr 2026

Volume 08 Issue 04 2026 | Crossref DOI: 10.37547/tajet/Volume08Issue04-16

Abstract

This article compares the MSVC and Clang/LLVM compiler stacks for native ARM64 builds in the Windows on Arm environment on Qualcomm Kryo and Oryon platforms. The study's relevance lies in the transition of the Windows ecosystem toward native ARM64 execution and the growing role of the compiler in determining software efficiency on new Snapdragon X systems. The aim of the study is to conduct a quantitative and qualitative assessment of code-generation differences between the two toolchains for computational workloads with different profiles. The novelty of the work is defined by a systematic cross-microarchitectural comparison of new MSVC and Clang/LLVM versions in the specific Windows on Arm environment, with separation of compiler influence from hardware platform influence. It was established that Clang/LLVM delivers higher performance in most tests in audio and video encoding, sorting, and interpreted code execution, whereas MSVC reveals a pronounced advantage in the isolated NumPy sqrt mathematical kernel on the Oryon architecture. The results confirm that code-generation efficiency depends on workload characteristics and processor microarchitecture. The article will be useful for developers of C/C++ systems, compiler researchers, and software architects targeting Windows on Arm.

Keywords: Windows on Arm, ARM64, MSVC, Clang, LLVM, compilers

Cite This Article: Khmyznikov, G. (2026). Comparative Analysis of MSVC and Clang/LLVM Compilation on Windows on Arm. The American Journal of Engineering and Technology, 8(4), 192–203. https://doi.org/10.37547/tajmei/Volume08Issue04-16

Introduction

Over the last decade, the ARM instruction set architecture has undergone a transformation, moving beyond mobile and embedded systems and securing a position in high-performance desktop and server computing (Rahman et al., 2024). The Windows on Arm operating system has faced performance problems due to the need to emulate x86/x64 code (Yen et al., 2025). Emulation, despite major improvements in instruction translators, introduces overhead in decoding and transforming architectural states, which constrains energy efficiency and peak performance (Xie et al., 2024). Since then, new hardware solutions have been introduced, such as the Snapdragon X family system-on-chip design with custom Oryon microarchitecture (Williams & Kanapathipillai, 2025), as well as Copilot+ PC class systems with a

specialized neural processing unit and integrated graphics (Yeverechyahu et al., 2024). In order to take advantage of the performance improvements brought by these hardware changes, the software industry is pushing for native ARM64 applications. Code that is compiled natively can take advantage of the wide pipelines, cache hierarchy, and wide vector math units provided in ARM processors (Poje et al., 2023). The quality of the compiled code is, therefore, important and the quality of code generation performed by the compiler is a key concern.

The C/C++ development environment for Windows on Arm currently relies on two distinct compiler stacks. The first is Microsoft Visual C++, or MSVC (Johnson, 2026). This is a proprietary, historically standardized tool for the Windows ecosystem, integrated into Visual Studio and the Windows SDK. Recent MSVC versions, including the v14.51 branch, incorporate important ARM64 backend revisions, the introduction of new loop optimization algorithms, improved scalarization, and expanded autovectorization support. The MSVC architecture is oriented toward seamless compatibility with proprietary OS mechanisms, such as structured exception handling and generation of debug information in PDB format. The second solution is the open LLVM infrastructure with the Clang frontend operating in clang-cl compatibility mode. LLVM has a mature AArch64 backend optimized over a decade by an industry consortium for Linux, Android, and macOS platforms, especially during the Apple Silicon transition (Bezbaruah et al., 2024) . The LLVM software pipeline implements a strict static single assignment and modular intermediate representation. Optimizations include loop vectorization and superword-level vectorization.

For developers and system architects, understanding the trade-offs in selecting a compiler toolchain is important. A central problem emerges: which compiler delivers the highest runtime performance for natively compiled ARM64 code, how the behavior of generated binaries differs across generations of ARM cores, and which concessions in seamless Windows integration must be made in exchange for superior code-generation quality. The purpose of this study is to conduct a quantitative and qualitative analysis of code-generation differences between MSVC and Clang/LLVM on the Windows ARM64 platform. Primary attention is given to the behavior of generated code as a function of specific computational workload profiles and Qualcomm Kryo hardware microarchitectures, in comparison with Qualcomm Oryon.

The scientific novelty of this work lies in systematic cross-microarchitectural testing of the latest compilers in the Windows on Arm environment, enabling isolation of the influence of SoC hardware improvements from the efficiency of code-generation algorithms. Most existing studies focus either on emulation versus native code or on ARM architecture testing in Linux server environments (Bezbaruah et al., 2024). This study closes that gap. Its practical significance is expressed in empirically grounded recommendations for compiler selection. Analysis of four open-source projects with different profiles allows us to draw conclusions applicable to industrial software development. Identification of unique microarchitecture-dependent exceptions, such as the NumPy kernel anomaly, opens new directions for research in low-level optimization for the Oryon architecture and revises the paradigm of unconditional superiority attributed to cross-platform translators. The study formulates two working hypotheses.

H1: In the Windows on Arm environment, the Clang/LLVM compiler stack provides higher performance of native ARM64 code than MSVC in most computational scenarios, including audio and video encoding, data sorting, and interpreted code execution.

H2: Compilation efficiency in Windows on Arm is determined by the interaction between workload profile and processor microarchitecture. As a result, in selected mathematical kernels on Qualcomm Oryon, MSVC may outperform

Clang/LLVM.

Materials and Methods

To reveal the influence of processor microarchitectural changes on compiled code performance, two hardware platforms were used, representing the previous and current generations of Qualcomm computing solutions for personal computers. Platform characteristics are recorded in Table 1.

Table 1. Hardware platforms and testing environment configuration

Parameter	Kryo Platform, Previous Generation	Oryon Platform, Current Generation
Device	Volterra / Windows Dev Kit 2023,	Microsoft Surface Laptop 15
SoC	Snapdragon 8cx Gen 3	Snapdragon X Elite, X1E80100
CPU Microarchitecture	8-core Kryo, based on ARM Cortex-X1 / A78	12-core Oryon, fully custom architecture
Operating System	Windows 11, build 10.0.27975	Windows 11, build 10.0.26595
OS Architecture	ARM64	ARM64

The Kryo cores used in Snapdragon 8cx Gen 3 are customized licensed ARM cores in a big.LITTLE configuration oriented toward a balance of energy efficiency and performance. In contrast, the Oryon cores in Snapdragon X Elite are based on a fully proprietary microarchitecture developed by Qualcomm. The Oryon architecture features a much wider frontend, an enlarged out-of-order execution window, improved branch prediction algorithms, and a massively redesigned cache hierarchy reaching up to 42 MB of shared cache (Williams & Kanapathipillai, 2025). These hardware differences imply different requirements for instruction scheduling on the compiler side.

]In the experiment, two current compiler stacks are compared using settings that ensure maximum performance on the target architecture. The configuration of the compiler toolchains is shown in Table 2.

Table 2. Configuration of compiler toolchains

Characteristic	MSVC	LLVM/Clang
Toolchain Version	v14.51, Visual Studio 2026 Preview	22.1.1, clang-cl mode
Base Optimization Flags	/O2 /GL /LTCG	-O3 -flto
Floating-Point Math	/fp:fast	-ffast-math
Target Architecture	ARM64	AArch64

An important methodological aspect requires emphasis. MSVC has no direct structural equivalent to the -O3 flag used in the LLVM ecosystem. The /O2 flag in MSVC expands into an aggregated set of speed-maximizing directives (/Og /Oi /Ot /Oy /Ob2 /Gs /GF /Gy), representing Microsoft's highest standard profile of optimization aggressiveness. In both LLVM and MSVC, the flags -flto and /GL /LTCG enable link-time optimizations, such as inter-procedural optimization for cross translation unit analysis of the global call graph and deep inlining at a whole program level. The flags /fp:fast and -ffast-math enable fast vector math. They relax IEEE-754 conformity by allowing the compiler to re-order algebraic operations, suppressing Not-a-Number and Infinity checks, and applying vectorized associative reductions that would not otherwise be algebraically correct (Tekriwal & Sarracino, 2025). These relaxed options are compared to the maximum MSVC optimization profile, and the aggressive LLVM profile.

The quality of this code generation was evaluated on four open-source software projects that represent different classes of computation.

LAME, an MP3 audio encoder, represents a mixed workload of digital signal processing. Its psychoacoustic modeling algorithms use dense loops with intensive integer arithmetic and floating-point operations. Such computational patterns are well-suited for baseline autovectorization and loop optimization.

NumPy, a library for matrix computation, represents the class of pure mathematical kernels associated with numerical array processing. In this case, two specific operations were analyzed, namely square root computation and array sorting. This workload enables evaluation of how effectively the compiler generates optimal SIMD instructions and manages memory bandwidth.

x264, an H.264 video encoder, is an resource-intensive task. It is characterized by high pipeline branching density, motion prediction, and sum-of-absolute-differences computation. This computational profile requires high-quality register scheduling and efficient use of vector units from the compiler.

CPython, the native Python interpreter, represents the class of virtual machines and general-purpose applications. Its key feature is a giant evaluation loop containing unpredictable indirect branches implemented via computed gotos or switch-case constructs. Performance in this case is determined by the degree to which instruction cache misses are minimized and by the quality of branch prediction heuristics.

To minimize result dispersion and exclude adverse effects from the OS thread scheduler, experimental controls were introduced. Executable processes were pinned to specific logical CPU cores to prevent thread migration, inter-core communication, and L1/L2 cache invalidation. Processes were assigned high priority class. Before the main measurements, a preliminary warm-up was performed to bring the processor out of power-saving states, stabilize clock frequencies, and populate the instruction cache. The number of runs varied with the duration of the specific workload, ensuring the required statistical power.

For LAME, 20 encoding runs were performed. For NumPy, 50 iterations of high-dimensional array processing were used. For x264, 3 full encoding passes of the reference video stream were conducted. For CPython, 15 independent subtests included in the CPU-bound benchmark suite were considered, each executed 40 times.

The process of project compilation and benchmark execution was fully automated using scripts. Raw metrics, namely execution time in seconds or milliseconds and throughput in frames per second, were collected and serialized into JSON format. Measured parameters included execution time, the relative advantage of one compiler over the other, and the sizes of generated binaries. For the CPython test set, the final metric was computed as the geometric mean across the entire benchmark

package to smooth out outliers. The source code of the testing infrastructure, build configurations, and the complete reproduction guide are recorded in the study repository (Khmyznikov, 2026).

Results

The collected data show that the LLVM/Clang toolchain demonstrates an overwhelming advantage in the absolute majority of tested scenarios. On both microarchitectures, code generated by LLVM ran faster, except for one specific mathematical kernel. The summary matrix of compiler performance on the Kryo and Oryon architectures is shown in Table 3.

Table 3. Compiler Performance Matrix Summary for Kryo and Oryon Architectures

Benchmark, Workload	Kryo Platform, MSVC	Kryo Platform, LLVM	Oryon Platform, MSVC	Oryon Platform, LLVM
LAME, Time, s, ↓	1.104	0.810	0.574	0.402
NumPy sqrt, Time, ms, ↓	4.322	4.209	0.629	2.113
NumPy sort, Time, ms, ↓	82.28	42.66	62.58	25.76
x264, Speed, fps, ↑	3.74	4.48	8.18	9.97
CPython, Geomean, ms, ↓	78.20	59.29	49.05	32.11

Note: The ↓ symbol means that a smaller value is better (execution time); ↑ means that a larger value is better (throughput).

The general conclusion from the presented data is that LLVM dominates across most computational paradigms. The result of the NumPy sqrt test on Oryon cores disrupts this congruence, demonstrating anomalous behavior of the MSVC code generator.

Testing of the LAME encoder confirms a stable and predictable LLVM advantage. On Kryo cores, LLVM code completes execution 26.6% faster, 0.810 s versus 1.104 s. On Oryon cores, the advantage rises to 30%, 0.402 s versus 0.574 s. This result is stable and free of dispersion outliers. In addition to timing characteristics, the compiler's influence on binary artifact size was analyzed, an important factor for embedded and resource-constrained systems. Comparative analysis of LAME encoding time is illustrated in Figure 1.

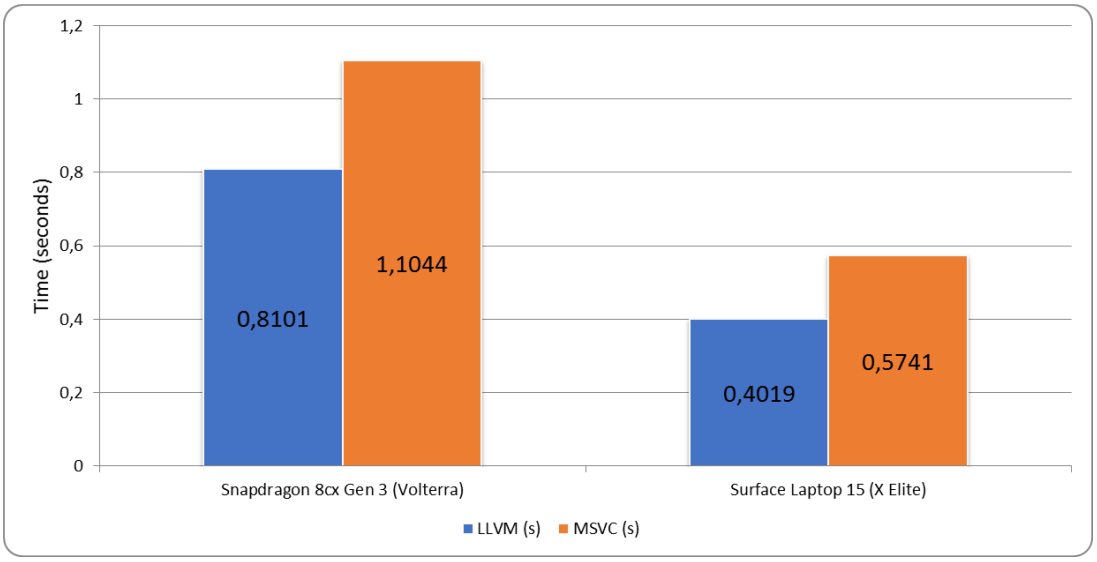

Fig. 1. Comparative Analysis of LAME Encoding Time

The size of the final executable file (.exe) proved comparable (MSVC: 513 KB, LLVM: 535 KB). Nevertheless, the size of the static library (.lib/.a) generated at link time differs radically: 4.0 MB for MSVC versus 1.3 MB for LLVM. This proves that the performance advantage of code does not necessarily coincide with an increase in final binary size. The substantial volume of MSVC files is due to how the intermediate representation is preserved in object files when the /GL flag is enabled.

Analysis of the NumPy subsystem is important because it is here that the microarchitectural dependence of optimizations is revealed.

Element-wise square root computation over a dense floating-point array demonstrates an unprecedented deviation. On the older Kryo platform, the difference between compilers is minimal and lies within the error margin, MSVC: 4.322 ms, LLVM: 4.209 ms. However, changing the hardware platform to Snapdragon X Elite yields a paradoxical result: MSVC completes the task in 0.629 ms, whereas the LLVM code takes 2.113 ms.

On Oryon cores, code generated by MSVC v14.51 is 3.4 times faster than code generated by LLVM 22.1.1. This proves a nonlinear compiler response to microarchitectural changes. Performance of NumPy sqrt operation on Kryo and Oryon architectures is shown in Figure 2.

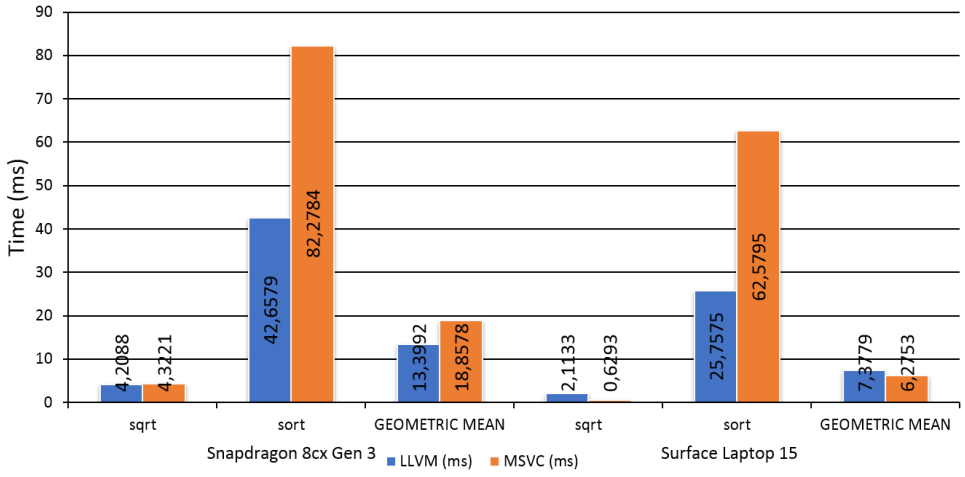

Fig. 2. Performance of NumPy sqrt operation on Kryo and Oryo architectures

By contrast, the sorting algorithm in the same NumPy framework paints a very different picture. In operations with intensive branching and memory rearrangements, LLVM significantly surpasses MSVC. On Kryo cores, LLVM is almost twice as fast, 42.66 ms versus 82.28 ms. On the new Oryon architecture, the gap remains and even increases slightly: LLVM completes sorting in 25.76 ms compared with 62.58 ms for MSVC. These two subtests, sqrt and sort, prove the thesis that translator superiority is not universal. It depends on the synergy between workload profile and processor microarchitecture.

Video encoding with the x264 library is a classic compute-intensive media workload. x264 algorithms heavily rely on macroblock computations, motion prediction, and integer SIMD vectorization.

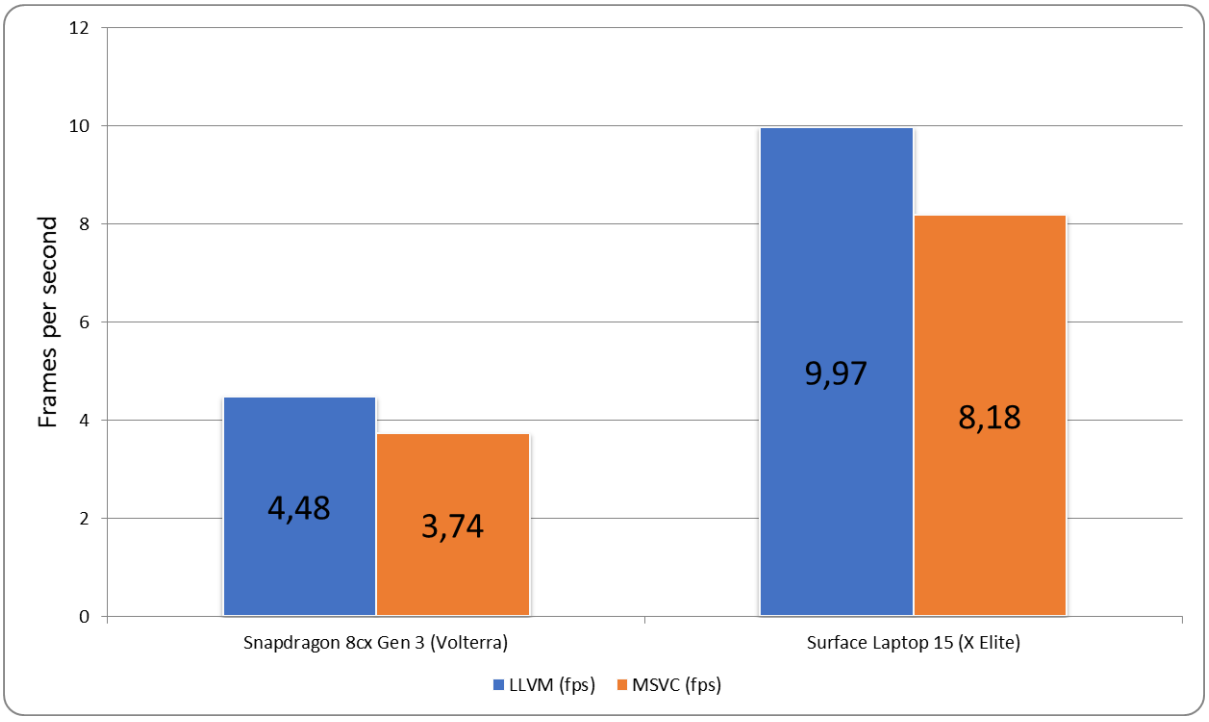

Fig. 3. Comparative Analysis of Encoding Throughput by FPS

On both hardware platforms, LLVM demonstrates a stable advantage. On Kryo, the gain is about 19.7%. On Oryon, it is about 21.8%. This confirms the overall strength of the LLVM code-generation pipeline for intensive integer media tasks.

The performance of virtual machines and interpreters depends on radically different metrics than those for mathematical kernels. In CPython, indirect branches, complex bytecode instruction dispatch logic, and interaction with the garbage collector dominate. According to the geometric mean across 15 CPU-bound tests, LLVM wins on both platforms. On Kryo, LLVM code executes in 59.29 ms, compared to 78.20 ms for MSVC. On Oryon, it executes in 32.11 ms versus 49.05 ms. CPython PyPerformance benchmark results are illustrated in Figures 4 and 5.

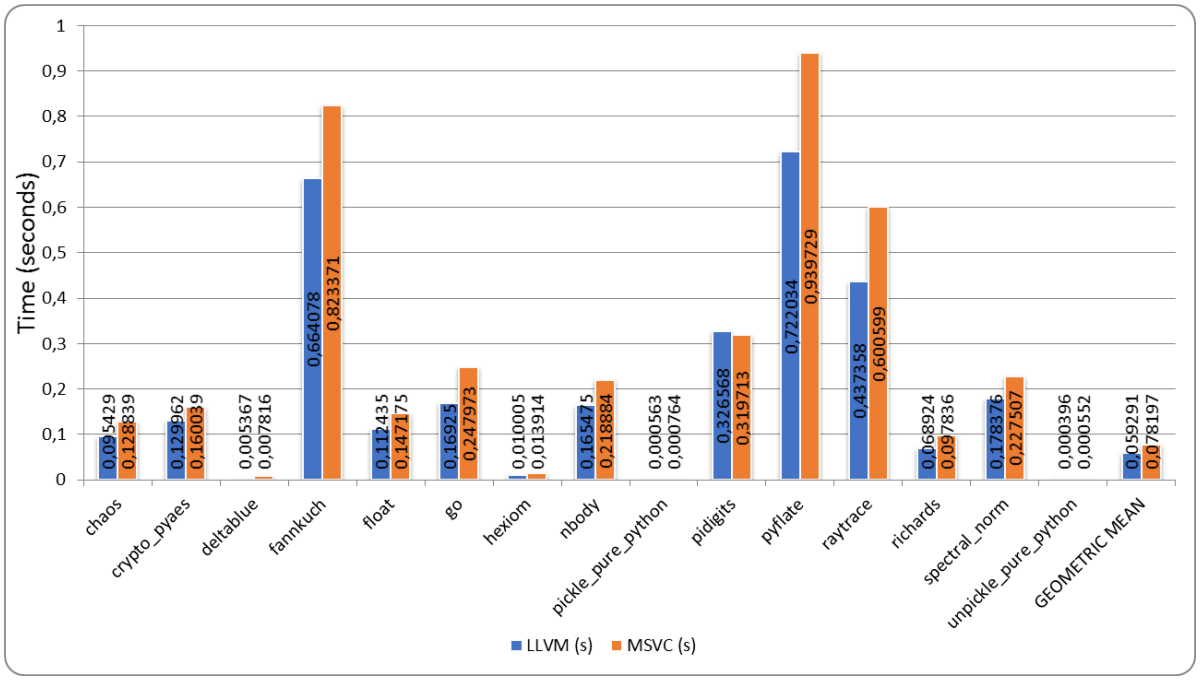

Fig. 4. CPython PyPerformance Benchmark Results on Snapdragon 8cx Gen 3

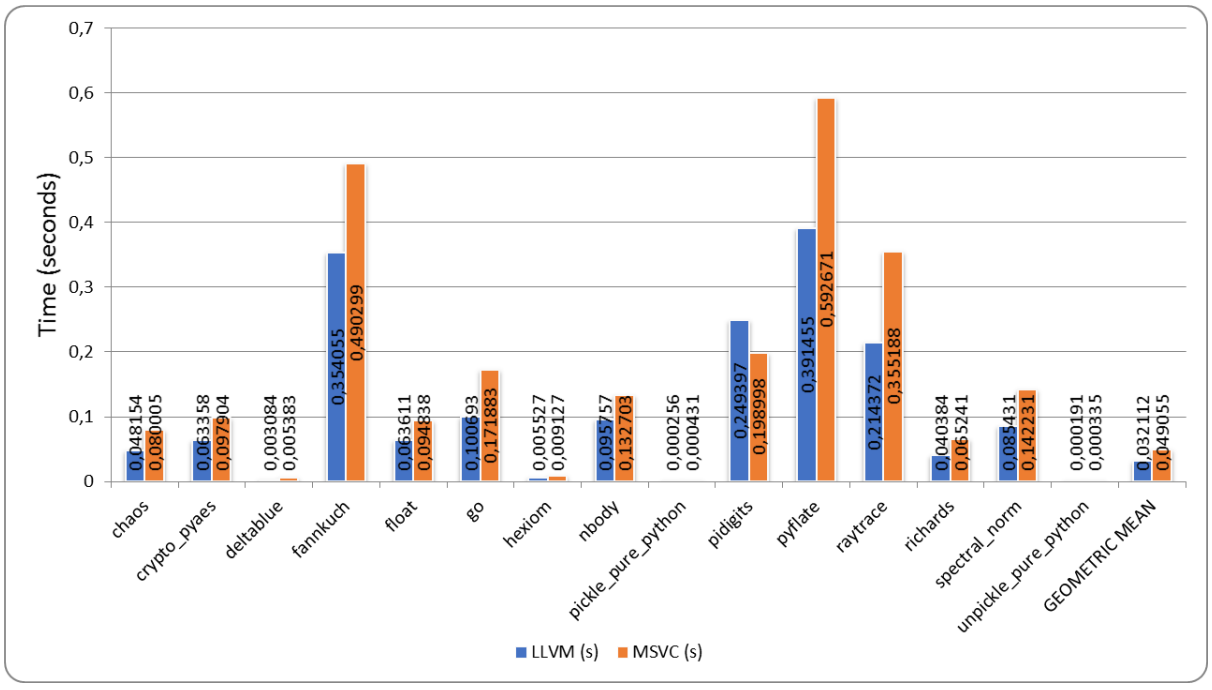

Fig. 5. CPython PyPerformance Benchmark Results on Surface Laptop 15

LLVM proved faster in 14 out of 15 tests and was only beaten in the pidigits benchmark, which calculates Pi and requires heavy use of arbitrary precision arithmetic. This benchmark provides the strongest evidence that LLVM has a superior architecture in dispatching a complex control-flow graph.

When comparing different translator types and processor generations, it appears that the code produced by LLVM outperforms the code produced by MSVC (Figure 6).

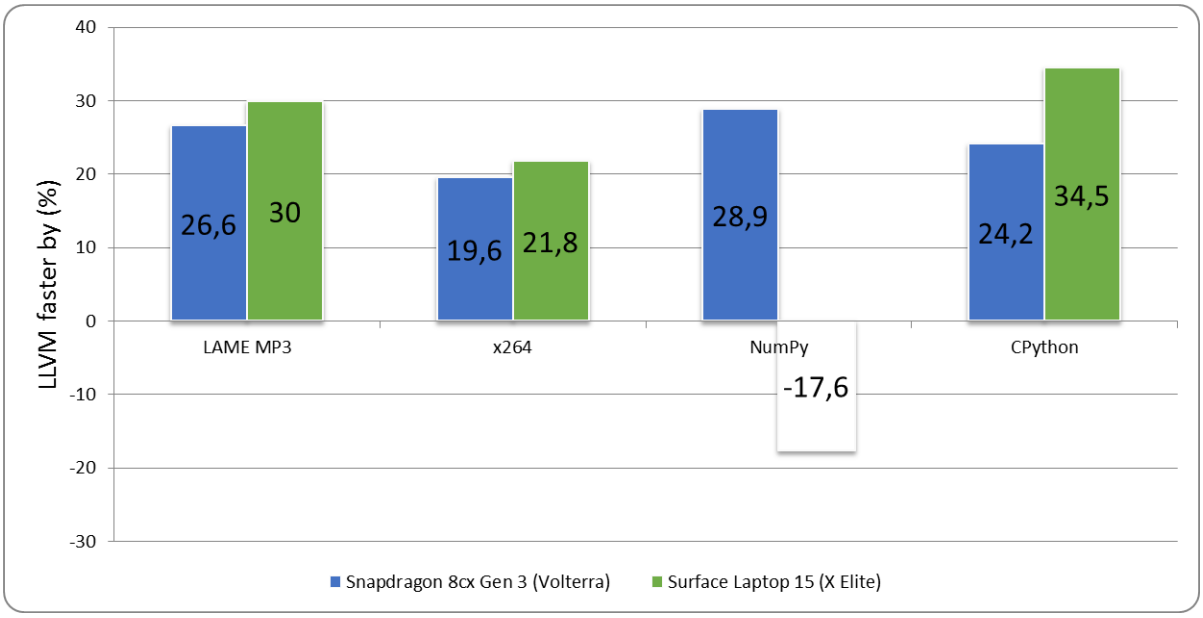

Fig. 6. LLVM Performance Advantage over MSVC

The transition to Snapdragon X Elite delivers a significant increase in IPC for both toolchains. However, the compiler response is not symmetric. In the LLVM case, performance in most tests' scales linearly with hardware improvements. At the same time, the MSVC compiler demonstrates the presence of special microarchitecture-specific execution paths, as in the NumPy sqrt case, which yielded no benefit on Kryo and produced an explosive effect on the Oryon architecture.

Discussion

The principal logic explaining LLVM's systemic leadership across most heterogeneous tasks, such as media encoding, virtual machines, and sorting, lies in its evolutionary heritage. The AArch64 backend in LLVM is one of the most mature in the industry. For many years, it has served as a primary instrument for the Linux, Android, and Apple Silicon ecosystems. Thousands of person-years of optimization have been devoted to improving the instruction selection framework, which evolved from SelectionDAG to the modular GlobalISel, enabling highly efficient machine code generation for RISC architectures.

LLVM also possesses a structural advantage in optimization aggressiveness. The absence in MSVC of a full equivalent to the -O3 flag means that the LLVM profile is initially oriented toward maximal performance extraction through code expansion, whereas MSVC /O2 attempts to preserve balance. LLVM has superior autovectorization heuristics, a loop vectorizer, and an SLP vectorizer, which transform scalar constructs into NEON/SVE instructions with greater efficiency. For projects such as CPython, the LLVM advantage stems from superior optimization of computed-goto constructs, which are important for the bytecode dispatcher.

The phenomenon observed in the NumPy sqrt operation on the Oryon architecture is also important. The 3.4-fold acceleration from MSVC on Snapdragon X Elite cannot be explained by a simple increase in core clock frequencies. This is evidence that the same compiler can exhibit radically different behavior across different generations of ARM64 cores.

Square root computation via fsqrt in the AArch64 architecture is an expensive operation with a latency of 7-32 cycles, depending on the processor's execution units (Moroz et al., 2021). When compiled with the fast-math flags /fp:fast or -ffast-math, the translator is entitled to replace the precise

hardware instruction with an approximation using the frsqrte instruction, followed by Newton-Raphson iterations employing fast fmul/fadd multiplication and addition instructions. The performance gap suggests that MSVC v14.51 includes hidden microarchitecture-dependent optimization paths that map with high precision to the wide out-of-order window of Oryon cores. LLVM, by contrast, likely generates vector or scalar code that encounters structural barriers in Oryon, such as execution port conflicts, delays in the memory subsystem, or memory backend stalls, resulting in pipeline stalls. The fact that this MSVC advantage is absent on Kryo cores confirms that Microsoft's code generator already possesses dedicated heuristics for the newest Qualcomm processors.

The transition from the theoretical science of performance to the realities of industrial development reveals substantial infrastructural barriers. Although clang-cl was designed as a drop-in replacement for cl.exe, providing ABI compatibility, in practice, building complex Windows projects with it can be difficult. Integration of a cross-platform compiler into the native Windows ecosystem exposes problems with LTO and the lld-link linker. Developers encounter incomplete support for MSVC-specific intrinsics in Clang, which requires manual rewriting of platform-dependent code sections. In addition, the deep integration of Windows applications with Side-by-Side manifests, debug symbol generation, and SEH mechanisms makes the migration process complex and labor-intensive for large codebases.

The analysis conducted allows for an applied interpretation for software developers. Applications of the Media Encoding and Compute-Heavy classes associated with streaming data processing, cryptography, and video and audio encoding, which include tasks similar to x264 and LAME, should be migrated to LLVM. This choice is determined by the fact that its vectorization and register allocation heuristics yield the greatest gains.

Enterprise software forms an additional workload class with direct relevance to Windows on Arm deployment. Database engines, web servers, and business logic services combine request parsing, memory allocation, synchronization, indexing, serialization, and deep branch-heavy execution paths. Their performance is shaped by instruction scheduling, cache residency, branch prediction, and the quality of generated synchronization code. In such systems, compiler choice can influence throughput, latency distribution, and CPU efficiency under sustained service load.

Machine learning workloads also belong to the practical scope of compiler selection on ARM64 systems. Frameworks such as TensorFlow, PyTorch, and ONNX Runtime execute tensor operations through dense numerical kernels, graph-level dispatch layers, operator fusion paths, and mixed scalar-vector code regions. Performance in these workloads depends on loop transformation, register allocation, vector instruction selection, and memory-access locality. These factors affect inference pipelines, matrix multiplication, convolution, normalization, and other CPU-executed model operations that remain relevant in production environments.

Developer tools constitute a further domain with clear significance for native ARM64 adoption. Build systems, language runtimes, compilers, package managers, and development environments execute heterogeneous workloads that include parsing, dependency-graph construction, file traversal, code generation, compression, and process orchestration. This workload mix places sustained pressure on integer execution, control-flow handling, memory traffic, and startup behavior. The present benchmark set does not model this category in full, yet the results obtained for CPython, sorting, and media-oriented pipelines support the expectation that compiler selection can affect the responsiveness and throughput of ARM64 developer environments.For large runtime environments, including Python and Ruby interpreters, V8-level JIT compilers, and complex business logic with high branching density, LLVM

is also the preferred solution. This is linked to higher-quality control-flow graph optimization.

The situation with computational mathematical kernels requires a more cautious approach. In applications that make intensive use of floating-point arithmetic and are oriented toward new architectures such as Qualcomm Oryon, unique microarchitectural exceptions in favor of MSVC are possible. For this reason, rejection of Microsoft's native tooling in such scenarios is permissible only after preliminary profiling.

When interpreting the results, several limitations of this study must be considered. First, all benchmarks under consideration are CPU-bound. Performance and optimization of heterogeneous computation using integrated GPUs or neural processors were not analyzed. Second, the study remained outside the scope of ARM64EC, Microsoft's hybrid ABI that permits mixing x64 and ARM64 code within a single address space, which is important for the real migration of large software systems. Third, testing did not include comparisons of Profile-Guided Optimization mechanisms, which can restructure hot execution paths based on dynamic analysis.

The systemic paradigm shift lies in the fact that the Windows on Arm platform has reached a high level of maturity. Compiler selection has ceased to be a question of compatibility and has moved into the domain of fine-grained performance tuning, which requires deep ARM64-specific benchmarking from the ecosystem.

Conclusion

The conducted study demonstrates that the Clang/LLVM infrastructure delivers higher code-generation quality for the ARM64 architecture in the Windows on Arm environment. Across most tested computational profiles, from audio and video encoding to bytecode interpretation, LLVM outperforms MSVC. Many years of adapting the AArch64 backend across the Linux and macOS ecosystems have made LLVM a highly powerful tool for optimizing scalar and vector instruction streams.

However, , not all code is faster with LLVM. Computation of the square root in the NumPy sqrt benchmark for the Snapdragon X Elite has the MSVC compiler for the same code 3.4 times faster than its competitor for the Oryon cores due to special microarchitectural heuristics implemented only in Microsoft compilers. Consequently, for performance-critical native ARM64 applications under Windows, the LLVM compiler should be considered a reliable default. The final decision, particularly for highly loaded mathematical kernels and software targeting new Qualcomm processors, should be made only after profiling and benchmarking of the specific workload on the target die.

The results of the study confirm both hypotheses. The first hypothesis is supported by the findings for LAME, x264, NumPy sort, and CPython, where Clang/LLVM demonstrated an advantage on both Kryo and Oryon platforms. This pattern is associated with stronger optimization maturity in the AArch64 backend, more effective vectorization, and higher code-generation quality in compute-intensive, control-flow-heavy workloads. The second hypothesis is also supported, since the observed differences between the compiler stacks depended on both workload type and processor microarchitecture. The NumPy sqrt benchmark provides the clearest example, as MSVC showed a marked advantage over Clang/LLVM on Oryon. This result indicates the presence of microarchitecture-sensitive code-generation behavior and shows that compiler selection for Windows on Arm should be based on the target application's performance profile and the target platform's characteristics.

The identified gaps and microarchitectural anomalies form a clear vector for future research. Here, the key point is to expand the benchmark set to evaluate the impact of profile-guided optimization technology on the code quality of MSVC and LLVM. An independent evaluation of the efficiency of the ARM64EC hybrid standard in

real applications with a high share of inherited x64 code is required. Finally, the adoption of scalable vector extensions in new compiler revisions and the integration of machine learning methods into code-generation pipelines will become the next frontier in optimizing computational systems on the ARM64 architecture.

Acknowledgements

The author is an employee of Microsoft. The opinions and conclusions expressed herein are those of the author and do not necessarily represent the official policy, endorsement, or position of Microsoft.

References

1. Bezbaruah, M., Dhakulkar, S., Pandey, P., P, H., Kumar, S. A., & Sudarsan, S. D. (2024). Comparative Analysis of GCC and LLVM for Performance Optimization on Aarch64. *Proceedings of 2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–6. https://doi.org/10.1109/hpec62836.2024.10938451
2. Johnson, T. (2026, March 4). *C++ Performance Improvements in MSVC Build Tools v14.51 - C++ Team Blog*. Dev Blogs. https://devblogs.microsoft.com/cppblog/c-performance-improvements-in-msvc-build-tools-v14-51/
3. Khmyznikov. (2026). *llvm-msvc-arm64-paper/results at main*. GitHub. https://github.com/khmyznikov/llvm-msvc-arm64-paper/tree/main/results
4. Moroz, L. V., Samotyy, V. V., & Horyachyy, O. Y. (2021). Modified Fast Inverse Square Root and Square Root Approximation Algorithms: The Method of Switching Magic Constants. *Computation*, *9*(2), 21. https://doi.org/10.3390/computation9020021
5. Poje, K., Brcic, M., Knezovic, J., & Kovac, M. (2023). First Steps towards Efficient Genome Assembly on ARM-Based HPC. *Electronics*, *13*(1), 39. https://doi.org/10.3390/electronics13010039
6. Rahman, T. N., Khan, N., & Zaman, Z. I. (2024). Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency. *World Journal of Advanced Research and Reviews*, *21*(1), 817–835. https://doi.org/10.30574/wjarr.2024.21.1.0017
7. Tekriwal, M., & Sarracino, J. (2025). Towards Verified Compilation of Floating-point Optimization in Scientific Computing Programs. *ArXiv*. https://doi.org/10.48550/arXiv.2509.09019
8. Williams, G., & Kanapathipillai, P. (2025). Qualcomm Oryon CPU in Snapdragon X Elite: Micro-Architecture and Design. *IEEE Micro*, *45*(3), 8–14. https://doi.org/10.1109/mm.2025.3568807
9. Xie, W., Luo, Q., Tian, X., Huang, J., & Qi, F. (2024). Performance Improvements via Peephole Optimization in Dynamic Binary Translation. *Electronics*, *13*(9), 1608. https://doi.org/10.3390/electronics13091608
10. Yen, J., Wang, J., Huang, Z., Wei, Z., Zhang, Z., Chen, C., Yu, S., Wang, Y., Wang, H., & Qi, Z. (2025). ARMing x86 Games: Accelerating Binary Translation Using Software-Only Validated Flag Speculation. *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 183–195. https://doi.org/10.1145/3711875.3729163
11. Yeverechyahu, D., Mayya, R., & Oestreicher-Singer, G. (2024). The Impact of Large Language Models on Open-source Innovation: Evidence from GitHub Copilot. *ArXiv*. https://doi.org/10.48550/arXiv.2409.08379