Comparative Analysis of GCC and LLVM for Performance Optimization on Aarch64

Mriganka Bezbaruah, Samruddhi Dhakulkar, Prachi Pandey, Haribabu P, S A Kumar, S D Sudarsan
Centre for Development of Advanced Computing, Bengaluru, India
{mriganka, samruddhi, prachip, hari, sakumar, sds}@cdac.in

Abstract— **The emergence of ARM-based architectures in high-performance computing (HPC) has necessitated a closer examination of compiler behaviors and optimizations. This paper investigates the performance and optimization capabilities of GCC and LLVM compilers for Aarch64 processors on HPC workloads. We focus on ARM-based HPC processors used in systems like Grace Hopper and Fujitsu A64FX, emphasizing on their implementations of the Scalable Vector Extension (SVE) and other architectural features that enhance performance. Through comprehensive benchmarking, including vectorization analysis with the Test Suite for Vectorizing Compilers (TSVC), we evaluate the performance of the compilers based on execution times, applied optimizations, and code portability across these systems. By comparing recent versions of GCC (v14) and LLVM (v18) with their previous versions, we highlight performance improvements and identify optimization gaps. Our analysis of assembly code offers insights into vectorization, register allocation, and SIMD instruction generation, providing valuable recommendations for future compiler enhancements. This study aims to give insights for the development of more efficient compilers, ultimately advancing the performance of Aarch64-based HPC systems.**

Keywords— Compilers, ARM, HPC, Vectorization, TSVC, GCC, LLVM, SVE

I. INTRODUCTION

The architecture of a computer system has significant impact on the design of compilers. A well-designed compiler can make a considerable difference in the performance of an application. For this, the compiler must be able to generate code that is efficient for the specific architecture. For example, a compiler for RISC (Reduced Instruction Set Computer) architecture will need to generate different code than a compiler for CISC (Complex Instruction Set Computer) architecture [1].

Traditionally, CISC machines have a more complex instruction set since a single instruction can involve multiple operations. Therefore, the burden of achieving the desired performance falls more on the high-level language program code, essentially through the compiler, than on the software. RISC machines on the other hand, usually perform only one operation within an instruction, which simplifies the implementation but shifts the burden from the hardware to the software. The compiler must also take into account the specific instructions that are available on the processor with respect to the architecture of the computer system. For example, a compiler for a processor that has a floating-point unit will need to generate code that uses the floating-point unit when possible. This is because the floating-point unit can perform floating-point operations more efficiently than the general-purpose registers on the processor. This makes compiler design and development more critical and complex that must take into account many factors, including the architecture of the computer system, the instructions that are available on the processor, and the specific needs of the application.

Until recently, the CISC architecture i.e., x86/x86_64 was the de facto standard for HPC [2] systems, and these systems outperformed the RISC-based systems. the HPC landscape has seen a significant shift with the introduction of ARM-based architectures, Aarch64. ARM processors [3] are renowned for their energy efficiency, cost-effectiveness, and increasingly competitive computational power, making them attractive alternatives for x86_64 HPC systems. This shift is exemplified by the deployment of ARM-based supercomputers like Fugaku [4], which demonstrated exceptional performance and energy efficiency. Another reason for this is the use of better and efficient compiler and other system software.

This paper focuses on the Compiler behavior and performance on prominent ARM-based processors such as Grace Hopper and A64FX [5], emphasizing their implementations of Scalable Vector Extension (SVE) and SVE2 [6][7]. Through comprehensive benchmarking, including vectorization analysis with the Test Suite for Vectorizing Compilers (TSVC) [8][9], we evaluate execution times, optimizations, and code portability across SVE and NEON. We compare the latest versions of GCC [10] and LLVM [11], and with their previous versions, highlighting the performance improvements and identifying optimization gaps.

The paper is structured as follows: Section II provides an overview of related work on compilers and the high-end ARM processors. Sections III and IV discuss the benchmarking and analyzes the behavior of the compiler on the Aarch64 in different HPC workloads. Section V presents optimization opportunities for GCC and LLVM. Finally, the paper concludes with a summary of our findings and insights which would help developers contribute to improving the performance further in these HPC systems.

II. RELATED WORK

While ARM architectures offer significant potential, realizing their benefits in HPC requires a deep understanding of how compilers optimize for these platforms. Compilers like GCC and LLVM must effectively exploit ARM's unique features, such as SVE and NEON (Advanced SIMD), to achieve optimal performance. This involves sophisticated optimizations in areas like vectorization, register allocation, and SIMD instruction generation.

Previous research has explored various aspects of ARM HPC performance, highlighting the need for efficient compiler optimizations and examined the performance

characteristics of these aarch64 processors in comparison to other architectures commonly used in HPC, such as x86_64. These studies [12][13][14] have highlighted the potential of ARM architectures in delivering competitive performance in HPC workloads. Notably, the Scalable Vector Extension (SVE) has been of significant interest. SVE provides scalable vector lengths, allowing for better utilization of vector processing capabilities. Previous research has explored the benefits of SVE instructions for improving performance in mathematical computation.

While existing research has made substantial contributions to understanding the new aarch64 processors and its implications for HPC systems, this paper aims to present new experimentation results and performance evaluation of the latest version of the top-tier open-source compilers like GCC and LLVM on these aarch64 systems and their capability for optimizing HPC workloads.

III. Experimental Setup

A. Target Compiler

We used two popular open-source compilers [15][16]: GCC v14.1 and LLVM v18.1. Both compilers have been around for many years and have been extensively developed, improved, and tested. In addition, GCC and LLVM are trustworthy and stable compilers that are known to manage complex codebase and produce effective machine code. The front-ends of GCC and LLVM share similar capabilities that allow them to read source code written in C, C++, Fortran, Java, and other programming languages to generate Intermediate Representation (IR), which is optimized by the target independent optimizer and pass it to the back-end. The back-end then translates the optimized IR into target-dependent assembly code.

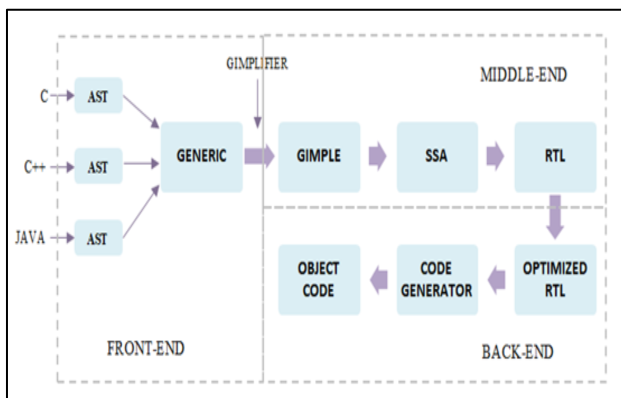

Fig. 1. GCC Structure

For GCC, the IR (known as GIMPLE) is passed multiple times through SSA, to produce RTL. The GIMPLE to RTL conversion involves generating RTL expressions, defining register assignments, and handling control flow. This is done by comparing the IR with the existing RTL templates by the GCC back-end. RTL again undergoes additional low-level optimizations, such as instruction scheduling, register allocation, and target-specific transformations to form optimized RTL which is fed into the machine code generation phase. LLVM on the other hand takes a more modular and customizable approach to code generation. It leverages a powerful intermediate representation (IR) called LLVM IR, which allows for more advanced optimizations and transformations. LLVM IR has its format in SSA with its own set of virtual registers. LLVM's code generation process, having LLVM IR as an input, involves a series of machine function passes, where each LLVM instruction transforms. After each transformation, the representation is nearer to the actual machine code. These passes work in a pipeline fashion, analyzing and transforming the LLVM IR to generate efficient assembly code.

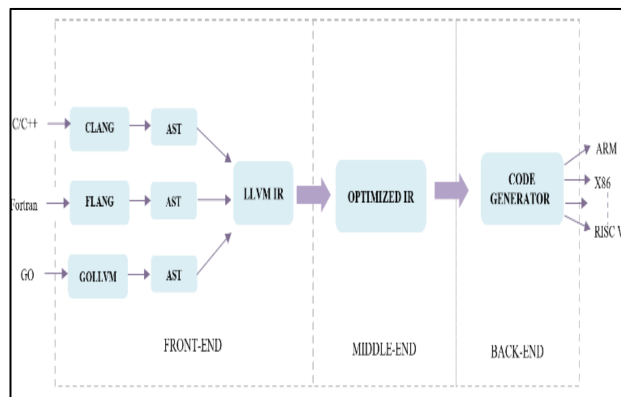

Fig. 2. LLVM Structure

One of the fundamental differences between GCC and LLVM is that GCC has a separate back-end for each target architecture, while LLVM has a single back-end that can be used for multiple architectures. This makes LLVM more flexible and easier to maintain, but it may also lead to suboptimal performance on some architectures.

Another difference between GCC and LLVM is the way they select instructions. While GCC uses a Pattern-matching approach for instruction selection while LLVM takes a table-driven approach based on a target description.

B. Target Machines

a) *Grace Hopper* [17]*:* The NVIDIA Grace CPU, based on ARMv9 architecture, features up to 148 cores running at high performance frequencies, optimized for HPC and AI workloads. It supports LPDDR5x memory with up to 1 TB/s bandwidth and includes private L1 caches for each core, shared L2 caches among clusters of cores, and a large shared L3 cache. With high-speed interconnects like NVIDIA NVLink, the Grace CPU ensures fast communication between CPUs and GPUs. Integrated with NVIDIA Hopper GPUs, it leverages Tensor cores and CUDA support for enhanced performance, making it good choice for supercomputing environments.

b) *Fujitsu A64FX:* It comes with ARM A64FX processor, HPC processor designed by Fujitsu that is based on the extended ARMv8.2-a architecture. The processors have 48 cores and can run at a base frequency of 1.8 GHz - 2.2 GHz depending on the implementation. TofuD or 100Gbps InfiniBand networking is utilized with a single socket set up for them. HBM2 memory, which was solely used on GPUs is now being used in this CPU, is organized in four stacks. Each stack has a direct connection to a Core Memory Group (CMG), which consists of 12 cores. Level 1 caches are private to each core, however Level 2 caches in a CMG are shared by all of the cores. Each CMG appears to the operating system as a separate NUMA node

C. Benchmark Selection

To assess the performance and optimization capabilities of GCC and LLVM for our target machines, we conducted benchmarking with some of the selected benchmarks. These benchmarks provide insights on various aspects of the compilers' performance, including their ability to handle various workloads and how they optimize code for specific computational tasks. Note that, here we have taken only single node performance of any given app into consideration and the results and observation could be different when executed in a multi-node cluster. When considering these results, it should also be noted that we have not optimized any of the benchmarks for the target system. We have implementation each of the given benchmark with OpenMP except GEMM.

GEMM focuses on matrix-matrix multiplication, a fundamental operation in many scientific and numerical computing applications. It is of time complexity O(n3). It tests the compilers' ability to optimize memory access patterns and how efficiently it utilizes the available hardware resources. When evaluating the performance of the compilers, we take into account the execution time (measured in seconds).

BabelStream [18], evaluates memory bandwidth and latency performance by measuring the throughput of different memory access patterns. It uses memory bandwidth bound kernels: Copy, Mul, Add, Traid and Dot. Using the best-performing iteration, the benchmark calculates the average bandwidth with respect to the array size of each input.

HPCG (High-Performance Conjugate Gradient) [19], tests the efficiency of sparse linear algebra computations and memory hierarchy usage. For HPCG, at least 1800 seconds (30 minutes) should pass throughout the official run.

MiniBude [20], is a molecular dynamics application that involves both floating-point computations and memory operations, requiring efficient handling of data dependencies and parallelism. It accepts an input deck, the number of poses, and the iteration as parameters. For our experimentation, we left everything as default, using the bm1 deck at 65536 poses for 8 iterations for the results to be consistent.

IV. Analysis of Compiler Behaviour

A. Assembly Code

We investigated the behaviour exhibited by the GCC and LLVM compilers, aiming to gain insights into their assembly code generation, compilation time, etc. with GEMM. The assembly code we considered is with the highest optimization level (-Ofast) and SVE is enabled for A64FX and Grace Hopper with the flags -march=armv8.2-a+sve and –march=native respectively for both GCC and LLVM. We used –msve-vector-bits=<n> for LLVM to set the SVE length because without this flag LLVM tend to use NEON. The argument value is 512 for A64FX and 256 for Grace Hopper.

Both the compiler generates assembly code that performs similar operations such as load, store, arithmetic, and control flow. However, there are differences in the assembly code size and the specific instructions used as we observed for GEMM. LLVM operates on independent modules. This modular design and the inclusion of additional metadata and information required for analysis and optimization purposes for each compiler unit, contribute to the increased code size in LLVM's assembly output.

LLVM's assembly code appears to utilize more vector registers (21 Z-registers) compared to GCC's code (3 Z-registers), potentially indicating different register allocation strategies. Using these SIMD registers, both the compiler decides to perform vectorization, but LLVM decides to perform multiply add vector and two separate fused multiply, indicating the use of loop unrolling to maximize efficiency in the computation. GCC on the other hand takes a different approach and decides to perform a single instance fused multiply-add, movprfx and floating point add vector operations. We could also see the use of inlining in LLVM, demonstrating more aggressive nature of optimization.

Additionally, The differences in control flow instructions (e.g., "whilelo", "b.any", “b.le” in GCC and "b.ne", "cmp", "b.eq", etc. in LLVM) are a result of variations in the compilers' control flow optimization strategies. These differences can be attributed to how GCC and LLVM handle loop optimizations, branch predictions, and control flow restructuring. The choice of optimization level can significantly impact the generated instructions, code size, and performance of the compiled code.

It is important to note that the size of the assembly code alone does not necessarily indicate the performance or efficiency of the generated code. The choice of instructions, their ordering, and the overall optimization strategies employed by the compilers play a significant role in determining the code’s quality and performance characteristics. The choice of optimization level can significantly impact the generated instructions, code size, and performance of the compiled code. Higher optimization levels tend to produce more complex and specialized instructions that exploit more advanced optimization techniques. For e.g., -O3 introduces more aggressive optimizations than -O2, including advanced loop transformations, inter-procedural optimizations, and automatic vectorization which aim to maximize performance but may increase the code size and compilation time.

B. Compilation Time

We captured the compilation time for different programs using both GCC and LLVM compilers. The results show that, in general, GCC exhibits longer compilation time compared to LLVM, indicating that GCC requires more time for performing optimizations. For GCC, Parsing and preprocessing constitute a significant portion of the compilation time (up to 57%). The optimization phase, including vectorization and loop transformations, takes again a substantial amount of time (43%). Despite these optimizations, there are gaps where GCC misses opportunities, primarily due to the complexity of certain functions and unpredictable memory access patterns. On the other hand, LLVM’s compilation time is primarily influenced by instruction selection, Clang front-end processes, and scheduling. The impact of automatic vectorization on the overall compilation time is relatively low for both compilers. Notably, the compilation time for GEMM with GCC shows fluctuations between multiple runs. Note that this performance is considered without enabling

any external projects of LLVM like Polly [21] which likely will increase the compilation time drastically.

Both GCC and LLVM employ various optimizations, including loop fission, loop fusion, loop unrolling, constant folding, and eliminating empty loops. These optimizations involve steps like Loop Recognizer, Loop Dependence Analysis, and Loop Optimizer. While these steps do not significantly increase the compilation time, they do impact the effectiveness and efficiency of the optimizations performed.

C. Porting of Executables

We attempted porting the SVE executables for GEMM and observed the interesting results. When running on the same machine, on Grace Hopper, GCC's performance for GEMM was both the best and most consistent on both machines. When compiling with -march=native -msve-vector-bits=<n> to generate SVE code in Clang, we observed that with n=256, the results were correct. With n=512, 1024, and 2048, the results were incorrect and as the vector size increased, the computations were increasingly skipped, resulting in a progressively lower resultant matrix value. On A64fx, the results were consistent for both GCC and LLVM. After porting, even when compiling with different vector sizes, the results remained consistent on Grace Hopper and took the same time as n=256. The compiler targeting A64FX architecture is producing more portable binaries, while accommodating the characteristics of SVE

D. Vectorization Analysis with TSVC

To know the vectorization characteristics of the compiler in a better way, we selected the TSVC benchmark. TSVC or Test suite for Vectorizing compiler is a benchmark suite designed to measure the performance of compilers and processors for vector and SIMD instructions. The benchmark suite contains a set of programs that make use of vector and SIMD instructions to perform various computational tasks. Table I contains of some of the results of the TSVC benchmark tried with GCC and LLVM with SVE enabled.

TABLE I. TSVC LOOPS SAMPLES WITH EXECUTION TIME

LOOPS	Grace Hopper		Fujitsu A64FX	
	GCC	*LLVM*	*GCC*	*LLVM*
s115	0.587	6.359	3.626	31.645
s118	1.804	0.388	15.552	13.009
s221	1.084	2.057	8.470	16.574
s231	0.049	4.158	0.432	41.105
s1232	0.749	2.916	29.789	24.468

[a] Time in seconds

The analysis of GCC and LLVM vectorization performance on Grace Hopper and Fujitsu A64FX architectures reveals distinct strengths and weaknesses of each compiler, influenced by the nature of the loops and the target architecture. On the Grace Hopper architecture, GCC generally excels in vectorizing simpler loop constructs. For instance, in the triangular saxpy loop (s115), GCC performs significantly better than LLVM, indicating its efficiency in handling memory dependencies and utilizing SIMD instructions effectively. Similarly, in partially recursive loops (s221), GCC shows superior performance, likely due to its effective loop distribution and vectorization of independent operations. However, GCC's performance is not universally superior; in the interchanging triangular loops (s1232), LLVM outperforms GCC, revealing LLVM's strength in optimizing more complex loop patterns on Grace Hopper.

On the other hand, LLVM demonstrates exceptional performance in specific scenarios on Grace Hopper. For example, in potential dot product recursion (s118), LLVM significantly outperforms GCC, suggesting that LLVM's optimization techniques handle recursive dependencies more efficiently. However, LLVM struggles with certain loop constructs, such as the loop interchange with data dependency (s231), where it performs poorly compared to GCC.

The performance trends differ on the Fujitsu A64FX architecture. Here, GCC generally shows better performance than LLVM for several loop constructs. For instance, in the triangular saxpy loop (s115) and loop interchange with data dependency (s231), GCC outperforms LLVM by a wide margin, indicating effective vectorization and optimization for these patterns. However, GCC faces significant challenges with more complex loops, such as the interchanging triangular loops (s1232), where its performance drops dramatically. Conversely, LLVM, despite its overall slower performance, handles this complex loop pattern better than GCC, suggesting its optimization strategies. Understanding these nuances is crucial for selecting the appropriate compiler and optimization strategies tailored to their specific workload and target architecture.

E. Performance Benchmarking

To ensure a fair comparison between GCC and LLVM on both the Grace Hopper and Fujitsu A64FX architectures, we meticulously configured the compilation and runtime environments. We enabled the -O3 and -Ofast optimization flags for both GCC and LLVM. The -Ofast flag, in particular, activates the -ffast-math flag, which allows a broad range of optimizations by relaxing strict adherence to the IEEE 754 standard for floating-point arithmetic. Auto-vectorization, crucial for ARM architecture, is enabled at -O2 or higher as of GCC v14.1. For targeting the SVE (Scalable Vector Extensions) on A64FX and Grace Hopper, we used the flags -march=armv8.2-a+sve -msve-vector-bits=512 for A64FX and -march=armv9-a+sve for Grace Hopper. In addition to compiler optimizations, we set several environment variables to control system behaviour and resource allocation. We configured large page settings with XOS_MMM_L_PAGING_POLICY=demand:demand:demand and set XOS_MMM_L_ARENA_LOCK_TYPE=0 to influence memory management. For OpenMP parallelization, we specified OMP_PROC_BIND=close and OMP_PLACES={0:47:1} to ensure efficient and similar thread placement and binding. Additionally, we also utilized all available cores on the Grace Hopper machine to gauge peak performance.

BabelStream: On A64FX, the best performance is seen with LLVM with SVE 647 GB/s which is 22% more than the peak performance of GCC. On Grace Hopper, both the compilers performed closely but LLVM outperformed GCC for Traid operation with a difference of about 50 GB/s. With SVE on both A64FX and Grace Hopper, GCC is outperformed by LLVM. The A64fx system shows a decrease in performance with SVE when using GCC, while the Grace Hopper system shows improvements with SVE for both compilers, with LLVM showing the most significant

improvement. In general, LLVM is seen to get benefitted from SVE than GCC on both systems. However, GCC outperforms LLVM without SVE on both systems.

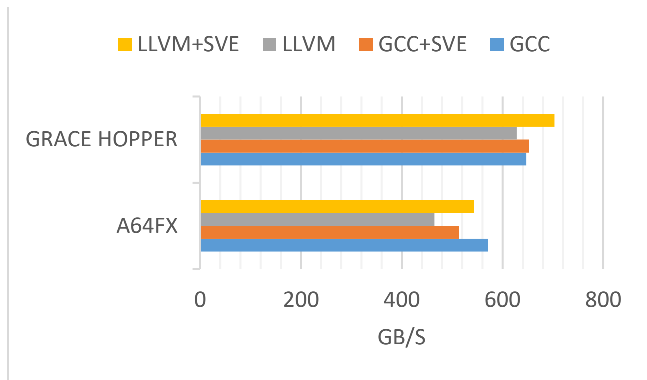

Fig. 3. BableStream

HPCG: LLVM consistently outperforms GCC in terms of both execution time and GFLOP/s, particularly when Scalable Vector Extension (SVE) is enabled. For instance, on the A64FX system, LLVM+SVE achieves 0.61 GFLOP/s for the smaller grid (128x128x128), compared to GCC's 0.36 GFLOP/s. Similarly, on the Grace Hopper system, LLVM reaches 4.05 GFLOP/s for the smaller grid, while GCC only achieves 2.85 GFLOP/s. The segmentation faults encountered on A64fx for larger grids needs further investigation, but this suggests a need for better memory handling and compatibility, particularly with larger datasets.

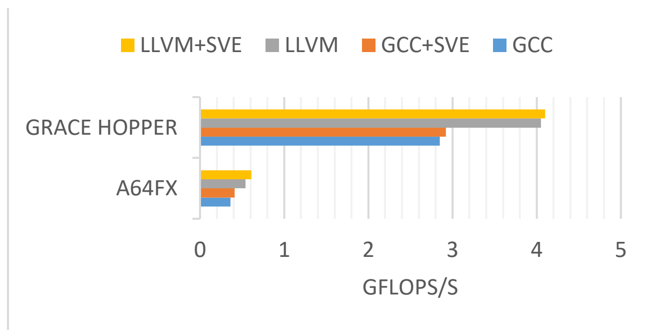

Fig. 4. HPCG

MiniBude: On the A64fx system, GCC outperforms LLVM by a significant margin. On the Grace Hopper system, LLVM outperforms GCC, especially when SVE is enabled, showcasing LLVM's superior optimization capabilities for this hardware.

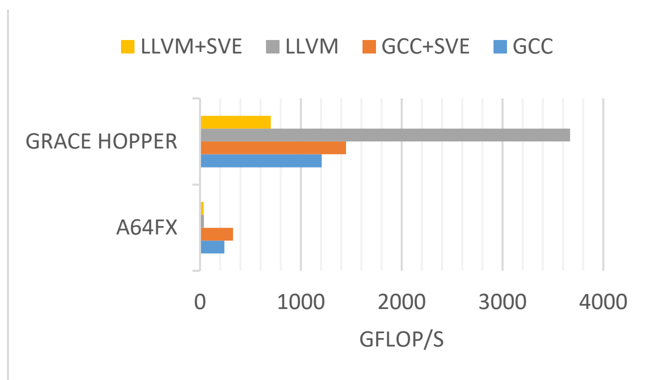

Fig. 5. MiniBude

The performance of GCC and LLVM across the three benchmarks exhibits some irregularities. These differences are influenced by several factors, including the nature of the benchmarks, the specific characteristics of the hardware, and the optimization capabilities of the compilers. This in turn affects the use of SVE registers and the efficiency of cache utilization resulting in varying performance results. LLVM consistently outperforms GCC in terms of both execution time and GFLOP/s, indicating better optimization capabilities for HPC workloads. The improvement in LLVM shows the benefit from enhanced vectorization and parallelism compared to GCC. Also, higher values of GFLOP/s signify more efficient use of computational resources, which is crucial for high-performance computing (HPC) workloads. While SVE enhances performance, GCC+SVE does not leverage this extension as effectively as LLVM does. GCC+SVE only marginally improves performance for most cases on the Grace Hopper system compared to LLVM.

V. Optimization Suggetions

While both GCC and LLVM provide a wide range of features and optimizations, there are notable opportunities for enhancing performance on specific targets like the A64FX processor. From our analysis, it can be concluded that GCC and LLVM exhibit distinct characteristics in terms of code generation and optimization, leading to varying performance outcomes. GCC needs to improve its support for SVE to fully exploit the hardware's capabilities. This could involve more sophisticated vectorization techniques and better alignment with ARM architecture specifics.

GCC provides a plugin framework that allows developers to extend the compiler's functionality and introduce custom optimizations. These plugins can perform additional analysis, transformations, or code generation techniques to enhance the optimization process. However, the scope of optimization that can be achieved through GCC plugins is limited compared to the core optimization features provided by the compiler itself. We can create Performance library plugins to optimize for specific computations which will give a significant performance boost. On the other hand, LLVM's extensive and powerful framework, with well-defined APIs, allows developers to implement major optimizations targeting specific architectural features as part of the compiler's optimization passes. Moreover, ongoing projects such as Polly and Vplan [22] can enhance LLVM's optimization capabilities by introducing advanced loop optimizations, vectorization techniques, and improving code generation. However, it should be noted that these external projects may sometimes introduce compatibility issues and other overheads.

Profiling indicated that GCC had significantly more branch instructions and L2 cache misses compared to LLVM. GCC was also observed to fail in loop unrolling when using SVE. Addressing these areas through more sophisticated optimization techniques can contribute to substantial performance gains. Additionally, using Profile guided optimization and improving vectorization strategies can further enhance the effectiveness of both compilers.

Beyond compiler optimizations, reviewing and optimizing the algorithms used in HPCG with the help of SVE intrinsics can yield significant performance gains. Although we experimented with various optimization flags,

there may still be combinations that could further maximize performance. By addressing these gaps and implementing these suggestions, both GCC and LLVM can enhance their optimization capabilities, leading to better performance for HPC workloads on aarch64 systems like A64FX and Grace Hopper.

VI. Conclusion

This study comprehensively evaluates GCC and LLVM compilers on ARM-based HPC architectures like A64FX and Grace Hopper, emphasizing their performance and optimization capabilities, particularly with respect to Scalable Vector Extension (SVE). LLVM consistently demonstrates superior performance metrics, achieving faster execution times and higher GFLOP/s rates compared to GCC across a range of benchmarks including GEMM, MiniBude, and HPCG. This superiority is particularly pronounced when SVE is effectively utilized, showcasing LLVM's adeptness in exploiting ARM's vector processing capabilities and architectural nuances. LLVM's assembly code analysis reveals sophisticated optimization strategies such as extensive loop unrolling, aggressive use of vector registers, and efficient memory access patterns, which contribute to its performance advantage. In contrast, GCC, while competitive in certain benchmarks like BabelStream on A64FX, exhibits longer compilation times and less optimized SVE utilization, indicating room for improvement in its handling of ARM-specific features. Future advancements could focus on enhancing GCC's SVE support and refining LLVM's memory management optimizations and vectorization to further elevate their efficiency in HPC applications on ARM architectures. This study will give valuable insights for developers and system architects aiming to maximize computational efficiency in ARM-based HPC environments, guiding future advancements in compiler technologies tailored to these platforms.

VII. Acknowledgement

I would like to thank Prof. Sameer Shende, University of Oregon for providing access to the Grace Hopper system on which we performed our experiments.

References

[1] Kirti Tokas, Dhruv Sharma, Lokesh Yadav, “RISC and CISC Architecture”, Internatonal Journal Of Innovative Research In Technology (IJIRT), 2014.

[2] A. Rico, J. A. Joao, C. Adeniyi-Jones and E. Van Hensbergen, ARM HPC Ecosystem and the Reemergence of Vectors, in Proceedings of the Computing Frontiers Conference, ser. CF’17, Siena, Italy: Association for Computing Machinery, 2017.

[3] Leonid Ryzhyk, “The ARM Archiytecture”, 2006.

[4] Ryohei Okazaki, Takekazu Tabata, and Sota Sakashita et al. Supercomputer fugaku cpu a64fx realizing high performance, high-density packaging, and low power consumption, 2020.

[5] Fujitsu A64FX, Online: https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/.

[6] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, et al., “The ARM scalable vector extension”, IEEE Micro, vol. 37, 2017.

[7] SVE2, Online: https://developer.arm.com/documentation/102340/0100/Introducing-SVE2

[8] Sergi Siso,Wes Armour, Jeyarajan Thiyagalingam, “Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information”, ACM Trans. Archit. Code Optim., Vol. 16, 2019.

[9] Saeed Maleki, Yaoqing Gao, and Tommy Wong et al., “An Evalution of Vectorizing Compilers”

[10] GCC, Online: https://gcc.gnu.org/.

[11] LLVM, https://llvm.org/docs/GettingStarted.html.

[12] Jens Domke, “A64FX – Your Compiler You Must Decide!” , IEEE International Conference on Cluster Computing (CLUSTER), Portland, Oregon, USA, 2021.

[13] Chanhyun Park, Misun Han, Hokyoon Lee, Myeongjin Cho, and Seon Wook Kim, “Performance Comparison between LLVM and GCC Compilers for the AE32000 Embedded Processor”, JEIE Transactions on Smart Processing and Computing, vol. 3, no. 2, April 2014.

[14] Adrian Jackson, Mich‘ele Weiland, and Nick Brown et al., “Investigating Applications on the a64fx”, IEEE International Conference on Cluster Computing (CLUSTER), 2020.

[15] GCC Internals, Online: https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture.

[16] Chris Lattner, “The Architecture of Open-Source Applications (Volume 1)”, Online: https://aosabook.org/en/v1/llvm.html.

[17] Wael Elwasif, William Godoy, Nick Hagerty, et. al, “Application Experiences on a GPU-Accelerated Arm-based HPC Testbed”, 2022.

[18] BabelStream, Online: http://uob-hpc.github.io/2021/12/22/babelstream-v40.html.

[19] HPCG, Online: https://github.com/hpcg-benchmark/hpcg/.

[20] MiniBude, Online: https://github.com/UoB-HPC/miniBUDE.

[21] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, “International Workshop on Polyhedral Compilation Techniques (IMPACT)”, 2011.

[22] Vectorization Plan [EB/OL], 2017, Online: https://llvm.org/docs/Proposals/VectorizationPlan.html.

[23] Jing Ge Fen, Ye Ping He and Qiu Ming Tao, “Evaluation of Compilers’ Capability of Automatic Vectorization Based on Source Code Analysis”, Hindawi, Scientific Programming Volume 2021.

[24] Top 500 June 2024, Online: https://top500.org/lists/top500/2024/06/

[25] Andrei Poenaru, Tom Deakin, and Simon N McIntosh-Smith et al., “An Evaluation of the Fujitsu a64fx for HPC Applications”, IEEE, Kobe, Japan, 2020.

[26] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions”, Phil.Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[27] NVIDIA Grace Hopper Online: https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper?ncid=no-ncid.

[28] Ezhil P and Ananthi Sheshasaayee, “Experimental Analysis of Optimization Flags in GCC”, Turkish Journal of Computer and Mathematics Education, 2021.

[29] A Performance-Based Comparison of C/C++ Compilers, Online: https://colfaxresearch.com/compiler-comparison/.