Time and energy consumption of multithreaded matrix factorization using various compilers optimizations

Beata Bylina[1,2][0000−0002−1327−9747], Monika Piekarz[1,3][0000−0002−3457−9335], and Jarosław Bylina[1,4][0000−0002−0319−2525]

[1] Maria Curie-Sklodowska University, Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin
[2] beata.bylina@mail.umcs.pl
[3] monika.piekarz@mail.umcs.pl
[4] jaroslaw.bylina@mail.umcs.pl

Abstract. This paper investigates the time and energy consumption of LU (with MKL library) and WZ multithreaded matrix factorization algorithms on Intel and AMD processors, utilizing OneAPI and Clang compilers. The study evaluates how processor architecture and compiler optimizations impact time and energy use during matrix factorization.
We describe the experimental setup, including hardware specifications, software configurations, and methods for collecting time and energy metrics. Both algorithms are tested under various conditions to assess their suitability for energy-efficient high-performance computing.
The results show variations in execution times and energy consumption based on the processor and compiler used. For LU factorization on Intel Xeon processors, Intel OneAPI optimizations prove most effective, while for WZ factorization on AMD EPYC processors, the Clang compiler demonstrates better performance. Choosing the right compiler options can reduce time and energy consumption by up to 6.5%.

Keywords: time · energy · WZ factorization · LU factorization · MKL · compilers options.

1 Introduction

Matrix factorization is a key concept in linear algebra, widely used in solving systems of linear equations. Among the most popular factorization methods are LU and WZ factorizations, which differ based on the specific application. LU factorization is a well-known factorization of a matrix into lower and upper triangular matrices, whereas WZ factorization offers an alternative approach that may yield better results on systems capable of parallel operations. A growing area of interest is the study of both algorithm execution time and energy consumption [3, 14] across different architectures [18]. Additionally, research on the impact of compilers on execution time and energy consumption [2, 10] is equally significant.

This paper aims to investigate the impact of compiler selection on execution time and energy consumption in multithreaded matrix factorization algorithms.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

In previous studies by the authors, published in the article [5], the focus was on how C/C++ compilers (GCC, ICC, OneAPI) affect performance and energy consumption in multithreaded WZ factorization on three computational platforms: two with Intel Xeon processors and one with an AMD EPYC processor. These studies revealed that compiler choice, combined with processor manual frequency scaling, has a significant impact on both performance and energy efficiency. The Intel compiler (OneAPI) achieved the best performance and energy savings in a multithreaded environment compared to the other compilers on each of the tested computing platforms.

In this paper, we extend our research to include LU factorization, enabling a comparison of both factorization methods across different compiler configurations. We implement LU factorization using the highly optimized Intel MKL library, which offers optimized multithreading, cache utilization, and vectorization, among other features. The study focuses on two compilers: Intel OneAPI, which achieved the best results in previous studies, and the Clang compiler, chosen for its increasing popularity and focus on code optimization. We consider various optimization levels available in both compilers, allowing for a comprehensive analysis of the impact of optimization on execution time and energy consumption in the context of multithreaded matrix factorization implementations using OpenMP. The experiments were conducted on two hardware platforms — one equipped with an Intel processor and the other with an AMD processor. Optimizing matrix factorization algorithms is crucial for improving the performance of numerous scientific applications and reducing their energy footprint. In particular, the efficient use of compiler optimizations can significantly accelerate computations in applications such as structural analysis, fluid dynamics modeling, and recommender systems.

The remainder of the paper is organized as follows. Section 2 provides a detailed description of the LU factorization algorithm implemented using LAPACK from the Intel MKL library, as well as the WZ factorization algorithm utilizing OpenMP-based parallelization with vectorization optimizations. Section 3 outlines the research methodology and presents an experimental analysis of the impact of various C/C++ compiler options on the performance and energy consumption of multithreaded matrix factorization (LU and WZ) on multicore architectures. Finally, Section 4 summarizes the findings and conclusions of the study.

2 LU and WZ factorization

Matrix factorization reduces a matrix to a product of two (or more) simpler matrices. This technique is often used as an auxiliary operation for solving linear systems, making subsequent systems easier to handle.

LU factorization transforms a square nonsingular matrix $\mathbf{A}$ into a product of two matrices: $\mathbf{A} = \mathbf{LU}$ where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. LU factorization without pivoting is possible when matrix $\mathbf{A}$ has a strictly dominant diagonal.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

In this article, we investigate the LAPACK [1] implementation of LU factorization without pivoting from the MKL library, specifically the `dgetrfnpi` function [8, 12]. These implementations are based on BLAS and optimized through the use of a multithreaded BLAS.

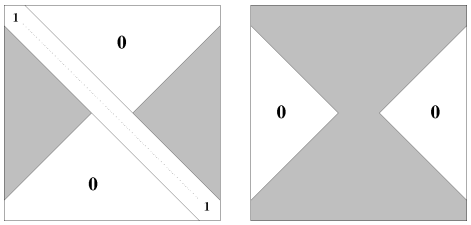

Fig. 1: The output of the WZ factorization — rows of the matrices **W** and **Z**

Next, we present the WZ factorization [9]. A square and nonsingular matrix **A** is decomposed into a product of two matrices, **W** and **Z**. The matrix **W** has a butterfly-like structure with ones on its main diagonal, while the matrix **Z** has an hourglass shape. These matrices are complementary in structure, meaning that one contains non-trivial elements where the other contains zeros or ones, and vice versa. The forms of these matrices are shown in Fig. 1.

```
for(k = 0; k < n/2-1; k++) {
    p = n-k-1;
    akk = a[k][k];     akp = a[k][p];
    apk = a[p][k];     app = a[p][p];
    detinv = 1 / (apk*akp - akk*app);
#pragma omp parallel for
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k]) * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p]) * detinv;
#pragma omp simd
        for(j = k+1; j < p; j++)
            a[i][j] = a[i][j]- w[i][k]*a[k][j]
                             - w[i][p]*a[p][j];
    }}
```

Fig. 2: The basic algorithm for the WZ factorization — pseudocode

Understanding the performance and energy efficiency of algorithms requires analyzing how they utilize computational resources. Table 1 presents this analysis for the LU and WZ algorithms, focusing on their memory access patterns, cache utilization, and the impact of vectorization and parallelism on floating-point operation performance [4].

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

Table 1: Comparison of LU and WZ algorithms

Characteristic	LU (with MKL)	WZ
vectorization	Full vectorization achieved through SIMD optimizations in MKL.	Semi-automated vectorization using `#pragma omp simd`; `-O3` flag enables automatic loop vectorization, but manual guidance improves control which we do not have in our case so there is no certainty of full vectorization.
parallelism	OpenMP; fully utilizes all available cores.	OpenMP; fully utilizes all available cores.
floating-point operations	$\frac{2}{3}n^3 + O(n^2)$	$\frac{2}{3}n^3 + O(n^2)$
memory access	The LU factorization primarily involves sequential access to matrix elements, which is generally efficient for memory systems.	The WZ factorization exhibits scattered memory access patterns, particularly when accessing elements of the W and Z matrices, which can lead to performance degradation.
cache utilization	Matrix blocking and data prefetching ensure high data locality, reducing cache misses.	Scattered memory access for elements w_{ik}, w_{ik2}, a_{ij}, a_{kj}, and a_{k2j} causes poor data locality, leading to frequent cache misses.

3 Numerical experiments

3.1 Methodology

Our experimental setup consists of two computing platforms: one with Intel Xeon processors and the other with AMD EPYC processors. Detailed specifications of these platforms, including clock speed, number of cores, cache size, and TDP (Thermal Design Power) for Intel Xeon Platinum and AMD EPYC processors, are provided in Table 2. These details are essential because they directly impact the performance and energy efficiency of the factorization algorithms. The AMD EPYC 9654 offers significantly more cores (192 vs. 64) and a higher maximum clock speed (3.7 GHz vs. 2.6 GHz) which can provide benefits for highly parallel workloads. However, the Intel processor has larger L1d and L3 caches (48 KB and 48 MB vs. 32 KB and 32 MB), potentially benefiting workloads that rely heavily on cache performance. Additionally, the L3 cache on AMD EPYC is modularly assigned to groups of 8 cores (per CCX), while on Intel Xeon, the L3 cache is shared among 32 cores, which may influence how memory-intensive workloads are handled across these architectures. These architectural differences are key to understanding the trade-offs between performance and energy efficiency in

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

the context of our study. A higher TDP signifies the ability to handle larger workloads but at the cost of increased power consumption. The choice depends on balancing performance and energy efficiency, as higher TDP processors excel in demanding tasks but incur greater operating costs.

Table 2: Specification of computing platforms

Processor	clock frequency [GHz]	cores	cache	TDP [W]
Intel Xeon Platinum 8358	0.8–2.6	2×32	L1i: 32KB L1d: 48KB L2: 1280KB L3: 48MB	270
AMD EPYC 9654	0.4–3.7	2×96	L1i: 32KB L1d: 32KB L2: 1024KB L3: 32MB	320

We ran each algorithm version five times at the maximum clock speed of the processors and averaged the results to ensure statistical validity. The Intel Xeon Platinum configuration used the full hardware performance with 64 threads, while the AMD EPYC processor was tested with 96 threads. For the considered problem size, 96 threads was sufficient, as adding more threads did not provide significant performance benefits. This is likely due to the relatively small problem size, which does not generate enough computational load to fully utilize the additional processing power of the AMD EPYC processor.

The dataset for our assessment comprises a randomly generated square matrix containing 32768×32768 double-precision values. Consequently, our test dataset encompasses 1073741824 cells, equivalent to 8GB of data. All algorithms adhere to a row-wise layout and are coded in C++, with vectorization and parallel processing.

Several C++ compilers are available for Intel Xeon and AMD EPYC processors, playing a key role in maximizing hardware potential. An efficient compiler abstracts low-level details while generating highly optimized machine code to enhance performance. However, finding one that consistently meets these expectations is challenging, as compilers may produce varying low-level instruction sets and differ in their ability to generate the most efficient code.

For our tests, we selected two widely available compilers optimized for our processors: Clang and Intel OneAPI. For the purposes of this article, the term 'OneAPI' will be used as a shorthand referring exclusively to the OneAPI compiler, and not to the entire Intel oneAPI toolkit. Both Intel OneAPI [11] and Clang [15] support modern C++ and OpenMP. However, OneAPI is specifically optimized for Intel processors, aggressively using features like AVX-512 and advanced prefetching for maximum performance. Clang prioritizes cross-platform compatibility, with less aggressive, more general optimizations for vectorization

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

(including AVX2/AVX-512) and memory management. Therefore, OneAPI may offer higher performance on Intel, while Clang provides greater versatility across different hardware.

In this study, we will explore various compiler options, selecting those supported by both Clang and Intel OneAPI to facilitate a fair comparison between the two compilers. Initially, we will compile with the OpenMP-enabled option along with the `-O3` flag, which directs the compiler to apply aggressive performance optimizations, including loop unrolling, vectorization, and other enhancements designed to improve program execution speed.

The following software was used during the tests along with the following compiler options:

- operating system: CentOS/Rocky/AlmaLinux 8.7
- kernel: Linux 4.18.0
- Intel OneAPI DPC++/C++ compiler v. 2022.0.0 with the following compiler options:
  `-qopenmp -O3`
- Clang v. 18.1.8 with the following compiler options:
  `-fopenmp -O3`

In further tests, in addition to enabling OpenMP support and the `-O3` flag, we will also evaluate the performance and energy efficiency impact of the following compiler options shown in Table 3.

Table 3: Compiler flags used for compilation (in addition to enabling OpenMP and the `-O3` flag, which is always included)

Case	OneAPI	Clang
I	[none]	[none]
II	`-funroll-loops`	`-funroll-loops`
III	`-march=native`	`-march=native`
IV	`-march=native -funroll-loops`	`-march=native -funroll-loops`
V	`-prof-gen=srcpos` `-prof-use`	`-fprofile-generate` `-fprofile-use=program.profdata`

Case I describes enabling only OpenMP and the `-O3` flag, without any additional options. The `-O3` flag enables aggressive optimizations, such as eliminating unnecessary instructions, optimizing loops, and using various other techniques to improve program performance. However, these optimizations can also increase compilation times.

Case II adds the `-funroll-loops` flag, which is used to unwind loops in the source code. This optimization can improve performance by reducing the number of loop iterations and improving data access patterns.

Case III adds the `-march=native` flag, which instructs the compiler to generate code optimized for the specific processor architecture on which the application will be executed. This allows the compiler to use processor-specific instructions and features, potentially leading to performance improvements.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

Case IV adds a combination of the `-march=native` and `-funroll-loops` flags. This dual approach optimizes the code for both the processor architecture and loop unwinding. The synergy between these options can result in significant performance improvements, especially for computationally intensive loops.

Case V includes the use of profiling-based optimization flags: `-prof-gen`/`-prof-use` for OneAPI and `-fprofile-generate`/`-fprofile-use` for Clang. The first step of this process generates profile data, while the second step uses this data to optimize code based on real-world execution metrics, ultimately improving performance in the final program.

We used the RAPL (Running Average Power Limit) interface [13,16] to measure CPU power and energy consumption, accessing its energy meters via Machine-Specific Registers (MSR). These 32-bit counters track energy usage since startup, updating every 1 ms. RAPL has been widely adopted for energy measurement and modeling, offering a practical alternative to complex power meters [13]. Prior studies [6,7,17] and our experience confirm its accuracy in measuring energy consumption for scientific applications.

To analyze performance and energy consumption differences across compiler configurations, we collect data on memory references, page faults, and branch instructions using `perf stat`. Examining memory access patterns, page faults, and branch behavior helps identify potential bottlenecks, inefficient memory usage, and control flow inefficiencies. This analysis provides insights into how compiler flags and processor configurations influence execution speed and power consumption, guiding optimization efforts.

3.2 The time and energy consumption for LU and WZ factorization algorithms

Based on Fig. 3 and Fig. 4, it is evident that the LU factorization algorithm utilizing the MKL library outperforms the WZ factorization algorithm in terms of both performance and energy efficiency. This is primarily because the WZ algorithm does not employ block mechanisms. The performance advantage of LU factorization is particularly pronounced on Intel processors, though still noticeable on AMD EPYC processors, albeit to a slightly lesser extent.

An analysis of LU and WZ factorization results on Intel Xeon and AMD EPYC processors also reveals significant differences in execution time and energy consumption between the Clang and OneAPI compilers. This section provides a detailed interpretation of these differences, highlighting the best cases for each processor, compiler, and algorithm.

LU factorization on Intel Xeon Platinum. The best performance and energy efficiency for LU factorization (Fig. 3) on the Intel Xeon processor were achieved using the OneAPI compiler with compilation I (baseline OpenMP compilation with `-O3`). This result stands out, as more advanced optimizations did not yield better performance, except for profiling (V), which produced similarly favorable results to case I. Clang also performed best in case I, although its performance degraded less than that of OneAPI when more

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

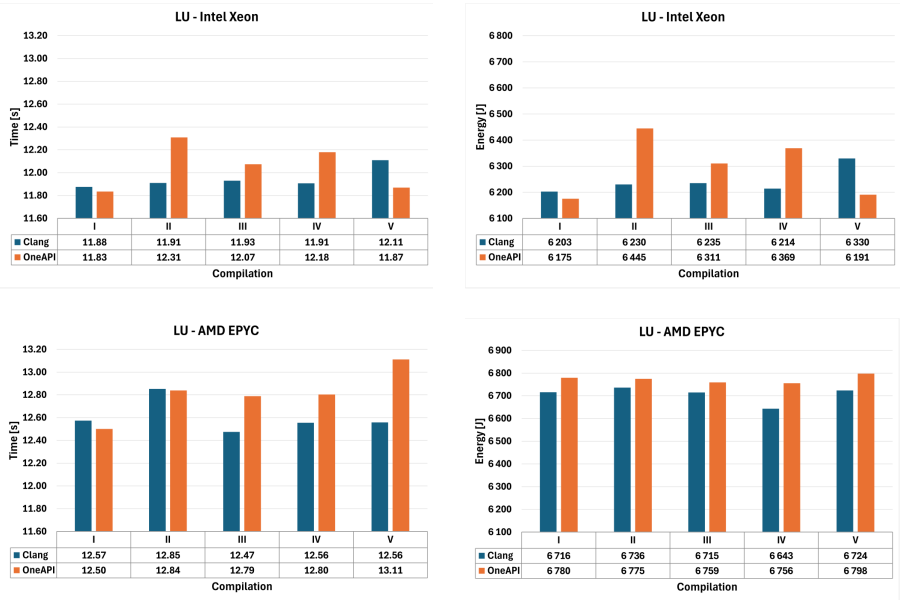

Fig. 3: LU — time and energy consumption on Intel Xeon Platinum and AMD EPYC (labels I–V as in Table 3)

advanced optimizations were applied. This may be attributed to the fact that LU factorization relies on the Intel MKL (Math Kernel Library), which is natively optimized for the OneAPI compiler and Intel Xeon architecture. MKL effectively utilizes low-level hardware optimizations, resulting in better resource utilization even with baseline compilation.

LU factorization on AMD EPYC. On AMD EPYC, Clang produced lower execution times and energy consumption than OneAPI. The lowest energy consumption was recorded for Clang with option IV (compilation with `-march=native` and `-funroll-loops`), while the best runtime for LU was achieved with Clang using option III (`-march=native`). AMD EPYC benefits more from optimizations tailored to its architecture (`-march=native`). By compiling code for the native architecture, Clang takes advantage of AMD-specific instructions, leading to improved runtime and energy efficiency.

WZ factorization on Intel Xeon Platinum. For WZ factorization on Intel Xeon (Fig. 4), OneAPI consistently outperformed Clang in terms of runtime and energy efficiency across all configurations (I–IV), except for profiling (V). The greatest advantage of OneAPI over Clang appeared in case IV (`-march=native` and `-funroll-loops`), where OneAPI achieved approximately 3% better performance and energy savings. The best performance and lowest energy consumption for WZ factorization on Intel Xeon were observed in case III using the OneAPI compiler, with an advantage of about 3.3% over case IV. Clang, however, generated the most energy-efficient code in case V

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

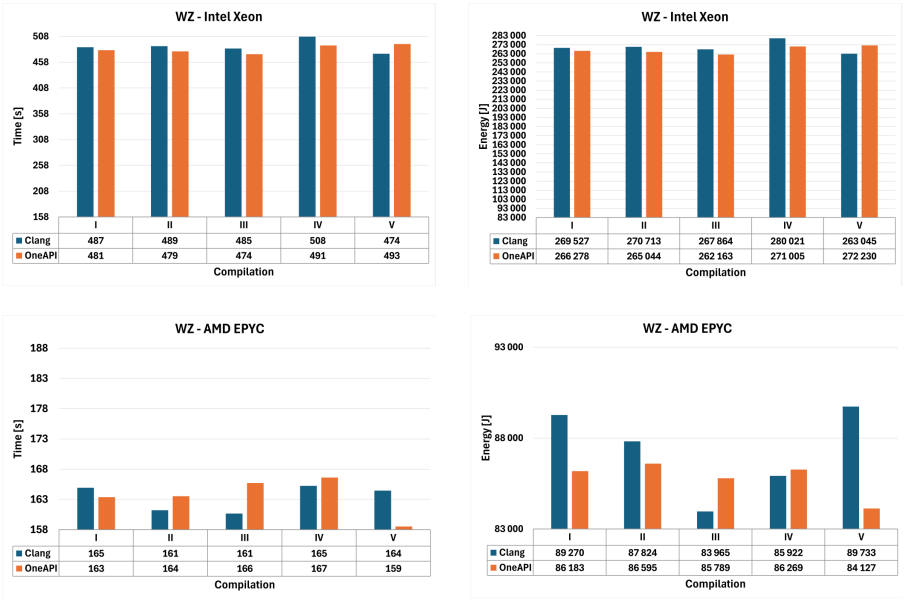

Fig. 4: WZ — time and energy consumption on Intel Xeon Platinum and AMD EPYC (labels I–V as in Table 3)

(profiling). The WZ algorithm is more sensitive to processor-specific instructions. On Intel Xeon, OneAPI manages performance more effectively during intensive matrix processing, while Clang produces more energy-efficient code when profiling is applied.

WZ factorization on AMD EPYC. WZ factorization is more efficient on AMD EPYC than on Intel Xeon (Fig. 4, bottom row), which features a different Y-axis scale for each processor (top and bottom rows in the figure). The differences between Clang and OneAPI in WZ factorization are less pronounced on AMD EPYC than on Intel Xeon. Clang slightly outperformed OneAPI in configurations III and IV, but with profiling (V), OneAPI performed better than Clang. This suggests that OneAPI's optimizations may be more effective for profiled code on AMD EPYC. For WZ factorization on AMD EPYC, the best runtime and lowest energy consumption were achieved using Clang with option III (`-march=native`). Comparably good energy results were obtained with the OneAPI compiler in the profiling case (V). WZ on AMD EPYC benefits from more aggressive loop optimizations and processor instruction tuning. Clang proves to be more efficient in both execution time and energy consumption when appropriate compiler flags are used.

3.3 Cache analysis

Cache misses play a crucial role in increasing runtime, as they often require fetching data from higher levels of the memory hierarchy or directly from RAM,

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

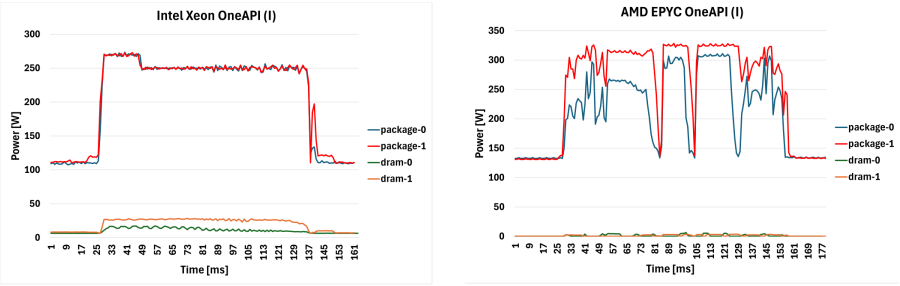

Fig. 5: Power profiling for LU factorization for the OneAPI compiler with the baseline option enabling OpenMP and the `-O3` flag. Left on Intel Xeon, right on AMD EPYC. Power shown is divided into processors' packages (*package-0*, *package-1*) and their respective memory (*dram-0*, *dram-1*)

causing delays that disrupt the processor pipeline and reduce computational efficiency. In addition to prolonging runtime, cache misses significantly affect energy consumption. While memory operations themselves consume relatively little energy, the processor's idle energy consumption during delays, such as waiting for data retrieval, is substantial (Fig. 5). In the idle state, power consumption ranges from 100 W to 150 W, with slightly lower values observed for the Intel Xeon processor compared to AMD EPYC. It accounts for over 40% of the total energy consumption during active computation. Although WZ factorization exhibits a lower percentage of L1 cache misses compared to LU, its overall performance is still heavily impacted due to the significantly higher number of memory references and absolute cache misses (Fig. 6).

As shown in Fig. 6, WZ factorization generates approximately:

- $10\times$ more L1 cache accesses than LU on Intel Xeon,
- $5\times$ more L1 cache accesses on AMD EPYC.

This increased memory load results in a higher absolute number of L1 cache misses:

- $8\times$ more L1 cache misses in WZ than in LU on Intel Xeon,
- $4\times$ more L1 cache misses in WZ than in LU on AMD EPYC.

This disparity leads to longer runtimes and higher energy consumption for WZ factorization.

The frequent cache loads observed in WZ factorization stem from the lack of blocking, which limits the reuse of data within the cache. As a result, the processor must repeatedly load new data into L1 cache, increasing the likelihood of evicting useful data and causing more misses. In contrast, LU factorization benefits from blocking, which improves data locality, increases cache utilization, and reduces the need for repeated memory accesses, leading to better performance and energy efficiency. Fig. 6 illustrates the different impacts of cache structure on LU and WZ factorization performance on Intel Xeon and AMD EPYC. LU

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

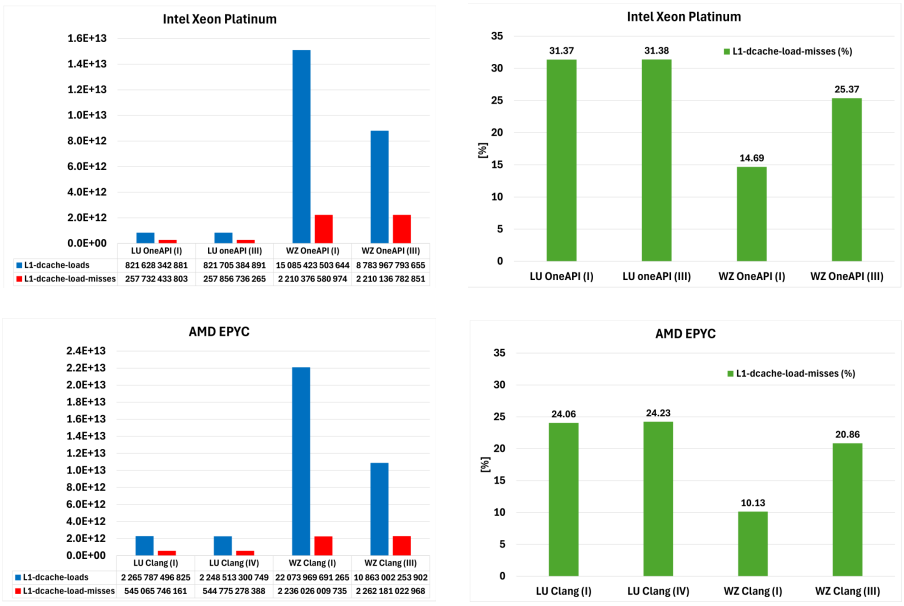

Fig. 6: Number of L1 data cache calls (blue), number of cache misses (red) and miss percentage (green). The best-performing compiler for each platform is shown with Case I and the top (or one of the top) configurations for each algorithm.

is relatively less affected by architectural differences between the two platforms, maintaining consistent performance and energy usage across both.

WZ factorization shows a significant improvement on AMD EPYC (approximately $3\times$ faster and $3\times$ lower energy consumption).We can observe that there are:

- 30% more L1 cache accesses on AMD EPYC compared to Intel Xeon (for OneAPI I),
- 18% more L1 cache accesses (for OneAPI III),
- 1–2% more cache misses.

Thus, we can see that the greater amount of cache access is not a bad thing, if only the number of cache misses is kept low.

This difference can largely be attributed to the modular structure of L3 cache in AMD EPYC, where smaller portions of L3 are assigned to groups of 8 cores (CCX). Unlike the centralized L3 cache in Intel Xeon, shared by 32 cores, AMD's modular design ensures more localized data access, reducing contention for cache resources and improving the efficiency of WZ factorization. Thanks to blocking, LU is less dependent on L3 cache, making it less sensitive to differences in cache architecture. These findings highlight that the L3 cache structure plays a crucial

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

role in the performance and energy efficiency of algorithms with high memory access variability, such as WZ.

Both Intel Xeon and AMD EPYC support AVX-512, but Intel Xeon benefits more from MKL optimizations, which are specifically tuned for Intel architectures. On AMD EPYC, the CCX modular structure and differences in cache affect performance.

We also investigated branch instructions in our algorithms. The differences between various compilation options for the same algorithm on the same platform are negligible — the only exception is LU factorization on AMD EPYC compiled by Clang (Fig. 7). The figure shows the impact of the compiler optimizations on the number of branches for this configuation. On AMD EPYC, loop unrolling (`-funroll-loops`) alone does not significantly reduce the number of branch instructions, but when combined with `-march=native` (compilation IV), there is a visible (about 5% less branches) improvement. This suggests that architecture-specific optimizations (`-march=native`) are key to fully leveraging the capabilities of AMD EPYC.

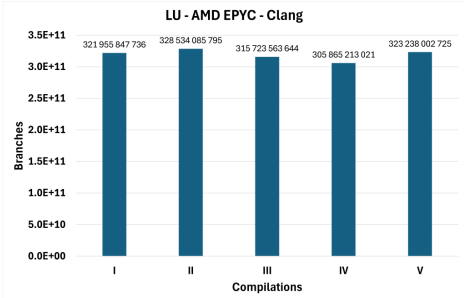

Fig. 7: Number of branches

Table 4 presents the best results achieved for LU and WZ factorization in terms of energy consumption. On the Intel Xeon platform, the best performance in both energy consumption and runtime was achieved for LU with OneAPI in compilation I, and for WZ with OneAPI in compilation III (`-march=native`), as shown in Fig. 3. In both cases, a lower number of L1 cache misses was observed compared to other compilations, as illustrated in Fig. 6.

4 Conclusions

The time and energy efficiency of LU and WZ factorization algorithms are highly dependent on the architectural platform and the choice of compiler. In the case of LU factorization, using Intel Math Kernel Library (MKL) optimizations, the Intel Xeon architecture showed better performance. In particular, the OneAPI compiler (Case I) achieved a 10% reduction in execution time and energy consumption compared to the AMD EPYC platform using OneAPI (Case V). In

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

Table 4: Best compilers and options cases in terms of energy consumption for LU and WZ factorization and particular platforms

Platform	Compiler	Case	Time [s]	Energy [J]	Performance [Gflops]	Energy efficiency [Gflops/J]
LU Factorization						
Intel Xeon Platinum	**OneAPI**	**I**	**11.83**	**6 175.13**	**1 982.07**	**3.80**
AMD EPYC	Clang	IV	12.47	6 598.84	1 880.37	3.55
WZ Factorization						
Intel Xeon Platinum	OneAPI	III	490.63	262 162.76	47.81	0.09
AMD EPYC	**Clang**	**III**	**160.67**	**83 964.60**	**145.99**	**0.28**

addition, tuning compiler options on AMD EPYC resulted in a 3.5% reduction in execution time (Case I).

WZ factorization, characterized by increased sensitivity to memory access patterns, shows a clear architectural dependency. The modular design of the AMD EPYC L3 cache effectively mitigates the impact of frequent L1 cache accesses, in contrast to the shared L3 cache structure on Intel Xeon, which can introduce performance bottlenecks. As a result, AMD EPYC with Clang (Case III) yielded a 68% execution time improvement and a 70% reduction in power consumption compared to Intel Xeon with Clang (Case IV). These observations are consistent with previous studies that address the critical role of compiler selection in optimizing performance and energy efficiency for multithreaded WZ factorization on similar architectures [5].

The optimal compiler configuration depends on the specific algorithm and processor architecture. Intel Xeon showed maximum benefit from basic OneAPI optimizations (Case I) for LU factorization, likely due to specialized MKL improvements. AMD EPYC, on the other hand, benefited more from aggressive architecture-specific compiler flags, with Clang (Case IV) providing better time and energy efficiency. The inherent dependence of WZ factorization on efficient memory access explains the consistent performance gains observed across architectures using the -march=native flag (Case III).

These differences are due to fundamental architectural differences and the different sensitivity of the algorithms to memory access and parallelization. The highly optimized MKL routines facilitated sufficient performance for LU factorization on Intel Xeon with basic flags, while AMD EPYC's distributed memory architecture required advanced tuning, such as the `-march=native` flag and loop unrolling. The reliance of WZ factorization on efficient memory access explains the consistent performance improvements in architectures with the `-march=native` flag (Case III).

Conversely, suboptimal compiler and flag choices for a given architecture can degrade performance. For LU factorization on Intel Xeon, using OneAPI with Case II flags, as opposed to Case I, resulted in a 4% reduction in execution time and energy efficiency. Similarly, on AMD EPYC, OneAPI (Case V) increased

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

execution time by 4% and energy consumption by 2% compared to Clang (Case IV).

In the context of WZ factorization, the choice of compiler remained significant. On Intel Xeon, Clang (Case IV) suffered a 6.5% loss in both runtime and energy efficiency compared to OneAPI (Case III). On AMD EPYC, Clang (Case V) showed a 2% increase in runtime and a 6.5% increase in energy consumption compared to Clang (Case III).

The optimal configurations across platforms and compilers are summarized in Table 4. For LU factorization, Intel Xeon with OneAPI (Case I) represented the optimal configuration, attributed to Intel-specific MKL optimizations. For WZ factorization, AMD EPYC with Clang (Case III) showed better performance, due to the lack of specific optimizations in our implementation, which highlights the importance of the choice of compiler and architecture.

Profile-guided optimization (Case V) offered additional benefits when other aggressive compiler options yielded diminishing returns. This strategy improved LU factorization on Intel Xeon with OneAPI, WZ factorization on AMD EPYC with OneAPI, and WZ factorization on Intel Xeon with Clang (Fig. 3 and Fig. 4). By analyzing detailed execution data, profile-guided optimization identified and fixed inefficiencies that standard compiler flags might otherwise miss.

These findings underscore the importance of tailoring compiler optimizations to a specific hardware architecture and algorithm in order to achieve optimal performance. Future research efforts could explore more advanced optimization techniques, extend the scope of numerical algorithms, and evaluate additional compiler suites. Furthermore, examining the impact of hybrid parallel programming models and distributed computing frameworks would provide a broader perspective on these observations.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. https://doi.org/10.1137/1.9781611971446.
2. D. BRANCO and P. R. HENRIQUES. Impact of gcc optimization levels in energy consumption during program execution. Acta Electrotechnica et Informatica, 16(1):20–26, 2016. https://doi.org/10.15546/aeei-2016-0004.
3. C. Buduleci, A. Gellert, A. andFlorea, and R. Bra. Architectural and technological approaches for efficient energy management in multicore processors. Computers, 13(4):84:1–21, 2024. https://doi.org/10.3390/computers13040084.
4. B. Bylina and J. Bylina. The parallel tiled wz factorization algorithm for multicore architectures. International Journal of Applied Mathematics and Computer Science, 29:407 – 419, 2019. https://doi.org/10.2478/amcs-2019-0030.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11

5. B. Bylina, M. Piekarz, and J. Bylina. Importance of c/c++ compiler choice for performance and energy consumption of multithreaded wz factorization. BULLETIN OF THE POLISH ACADEMY OF SCIENCES TECHNICAL SCIENCES, 2025. https://doi.org/10.24425/bpasts.2025.153226.
6. B. Bylina, J. Potiopa, M. Klisowski, and J. Bylina. The impact of vectorization and parallelization of the slope algorithm on performance and energy efficiency on multi-core architecture. Annals of Computer Science and Information Systems, 25:2283–2290, 2021. https://doi.org/0.15439/2021F68.
7. J. Bylina, B. Bylina, and M. Piekarz. Influence of loop transformations on performance and energy consumption of the multithreded wz factorization. In Preproceedings of the of the 17th Conference on Computer Science and Intelligence Systems, page 479–488, 2022. https://doi.org/10.15439/2022F251.
8. J. W. Demmel. Applied Numerical Linear Algebra. SIAM, 1997.
9. D.J. Evans and M. Hatzopoulos. A parallel linear system solver. International Journal of Computer Mathematics, 7(3):227–238, 1979.
10. K. Halbiniak, R. Wyrzykowski, L. Szustak, A. Kulawik, N. Meyer, and P. Gepner. Performance exploration of various C/C++ compilers for AMD EPYC processors in numerical modeling of solidification. Advances in Engineering Software, 166:1–14, 2022. https://doi.org/10.1016/j.advengsoft.2021.103078.
11. Intel Corporation. Intel oneAPI Toolkits. https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html, n.d.
12. Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/, 2014.
13. K. Khan, M. Hirki, T. Niemi, J. Nurminen, and Z. Ou. RAPL in action: Experiences in using RAPL for power measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS), 3, 01 2018. https://doi.org/10.1145/3177754.
14. S. Khokhriakov, R. R. Manumachu, and A. Lastovetsky. Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance. Applied Energy, 268:1–18, 2020. https://doi.org/10.1016/j.apenergy.2020.114957.
15. LLVM admin team. Clang: a C language family frontend for LLVM. https://clang.llvm.org/, n.d.
16. G. Raffin and D. Trystram. Dissecting the software-based measurement of cpu energy consumption: a comparative analysis. IEEE Transactions on Parallel and Distributed Systems, 36:96–107, 2024. https://doi.org/10.1109/TPDS.2024.3492336.
17. L. Szustak, R. Wyrzykowski, T. Olas, and V. Mele. Correlation of performance optimizations and energy consumption for stencil-based application on Intel Xeon scalable processors. IEEE Transactions on Parallel and Distributed Systems, 31(11):2582–2593, 2020. https://doi.org/10.1109/TPDS.2020.2996314.
18. M. A. Zeroual, K. Isaieva, P. A. Vuissoz, and F. Odille. Performance study of an mri motion-compensated reconstruction program on intel cpus, amd epyc cpus, and nvidia gpus. Applied Sciences, 14(21):9663:1–25, 2024. https://doi.org/10.3390/app14219663.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97570-7_11