

Article

Energy–Performance Trade-Offs of LU Matrix Decomposition in Java Across Heterogeneous Hardware and Operating Systems

Francisco J. Rosa , Juan Carlos de la Torre * , José M. Aragón-Jurado , Alberto Valderas-González 
and Bernabé Dorronsoro 

Department of Computer Engineering, University of Cádiz, 11003 Cádiz, Spain;
franciscojavier.rosavega@mail.uca.es (F.J.R.); josemiguel.aragon@uca.es (J.M.A.-J.);
alberto.valderasgonzalez@mail.uca.es (A.V.-G.); bernabe.dorronsoro@uca.es (B.D.)

* Correspondence: juan.detorre@uca.es

Abstract

The increasing core counts and architectural heterogeneity of modern processors make performance optimization insufficient if energy consumption is not simultaneously considered. By providing a novel characterization of how the interaction between hybrid architectures and system software disrupts the traditional correlation between execution speed and energy efficiency, this research study analyzes the performance–energy trade-offs of parallel LU matrix decomposition algorithms implemented in Java, focusing on the Crout and Doolittle variants. This study is conducted on four different platforms, including ARM-based, Hybrid x86, and many-core accelerators. Execution time and speedup are evaluated for varying thread counts, while energy consumption is measured externally to capture whole-system energy usage. Experimental results show that the configuration yielding the maximum speedup does not necessarily minimize energy consumption. While x86 systems showed energy savings exceeding 80% under optimal parallel configurations, the ARM-based platform required distinct thread counts to minimize energy consumption compared with maximizing speed. These findings demonstrate that energy-efficient configurations represent a distinct optimization space that often contradicts traditional performance metrics. In the era of hybrid computing, green software optimization must transition from a simplistic “race-to-sleep” paradigm toward sophisticated, architecture-aware strategies that account for the specific power profiles of heterogeneous cores to achieve truly sustainable high-performance computing.

Keywords: LU decomposition; performance–energy trade-off; Multithreaded Java; hybrid architectures; Green Computing



Academic Editor: Rocco Furferi

Received: 19 December 2025

Revised: 14 January 2026

Accepted: 16 January 2026

Published: 19 January 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Matrices are fundamental tools in modern science, as they provide a compact and powerful way to represent and manipulate complex data. In computer science in particular, matrices are essential to tasks such as image processing, machine learning, graph algorithms, and computer graphics. For instance, neural networks rely heavily on matrix representations to model weights and activations, while graphics pipelines use matrices to perform geometric transformations efficiently. By structuring data and operations in a consistent mathematical framework, matrices enable algorithms to scale and generalize across a wide range of scientific and engineering problems [1].

However, as matrix sizes grow, the computational cost of operating on them increases rapidly [2]. Common operations such as matrix multiplication, inversion, and solving linear systems can become extremely expensive in both time and memory, especially for large-scale applications like big data analytics and scientific simulations. The sheer number of arithmetic operations required for large matrices can lead to performance bottlenecks, making naive implementations impractical for real-time systems or high-performance computing environments. This challenge drives the need for more efficient numerical methods and optimized algorithms.

One such method is LU decomposition, which factors a matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U). This decomposition significantly improves computational performance when solving systems of linear equations. By transforming the original problem into simpler triangular systems, it allows solutions to be computed efficiently through forward and backward substitution, rather than repeated costly direct methods. LU decomposition reduces overall computation time and improves numerical stability, making it a cornerstone technique in efficient scientific computing, while also enabling high-performance implementations that benefit from decades of optimization research in the field [3].

Concurrent with algorithmic advancements, the hardware landscape is undergoing a paradigm shift. While traditional processor design prioritized increasing the count of identical cores with predictable scaling, the industry has pivoted towards heterogeneous architectures. This trend is most visible in modern hybrid designs that integrate high-performance cores (P-cores) with energy-efficient cores (E-cores). These architectures decouple the direct relationship between core count and throughput by introducing processing units with distinct microarchitectures, instruction-level capabilities, and power profiles. Consequently, algorithms that were originally optimized for homogeneous multicore environments require a fundamental re-evaluation to determine their behavioral efficiency on this emerging class of hardware [4].

To fully assess the implications of this architectural movement, it is no longer sufficient to evaluate algorithmic performance solely in terms of execution time [5]. The coexistence of performance-oriented and energy-efficient cores in these modern processors introduces new trade-offs between speed and power usage that directly impact overall system efficiency [6]. An algorithm may achieve lower execution times yet does so at the cost of significantly higher energy consumption or, conversely, utilizes energy-efficient cores to reduce power usage while increasing runtime. Therefore, studying energy consumption alongside execution time is necessary to obtain a deep understanding of how algorithms such as LU decomposition behave on different architectures. Only by analyzing both dimensions together, it is possible to capture the true performance–efficiency relationship and determine whether modern processors deliver meaningful gains over their predecessors in practical, real-world scenarios.

In this work, we address these challenges by analyzing parallel LU decomposition algorithms implemented in Java. We focus on the Crout [7] and Doolittle [8] methods, evaluating their behavior across different architectures, ranging from contemporary heterogeneous processors to traditional homogeneous multicore systems and even high-core-count server-class systems. This evaluation also spans three different operating systems to provide a comprehensive view of the software–hardware interaction stack. This study, subsequently, aims to tackle the following research questions:

- RQ1. How do heterogeneous architectures affect the runtime of the considered parallel LU decomposition algorithms?**
- RQ2. How do heterogeneous architectures affect the energy consumption of the considered parallel LU decomposition algorithms?**

RQ3. What is the relationship between runtime and energy consumption for the considered parallel LU decomposition on heterogeneous architectures?

This work addresses the challenges mentioned above by analyzing parallel LU decomposition algorithms implemented in Java. The main contributions of this paper are summarized as follows:

1. A systematic evaluation of the Crout and Doolittle parallel LU variants, characterizing their scalability and runtime behavior in modern managed-language environments.
2. A rigorous experimental framework spanning four architectural paradigms (Hybrid ARM, Hybrid x86, Homogeneous x86, and Many-core x86), enabling a comprehensive comparison across heterogeneous platforms.
3. A high-precision, whole-system energy measurement approach that captures real-world power consumption beyond what internal software sensors can provide.
4. An empirical identification of architecture-specific energy–performance “sweet spots”, showing that energy-optimal configurations often differ from peak-performance settings in hybrid processors.

The remainder of this paper is structured as follows: Section 2 reviews the state of the art on LU decomposition and parallel computing in homogeneous and heterogeneous architectures. After that, Section 3 describes the methodological framework, including the LU algorithms, measurement protocols, statistical tests, and energy–performance metrics, while Section 4 details the experimental setup, covering the evaluated algorithms, hardware and software configurations, and energy measurement instrumentation. Then, Section 5 presents the results on execution time and energy consumption, which are analyzed and discussed in Section 6 with a focus on performance–energy trade-offs. Finally, Section 7 addresses threats to validity, and Section 8 concludes the paper and outlines directions for future research.

2. Related Work

2.1. Algorithmic Foundations of LU Decomposition

In mathematics, dense linear algebra operations rely heavily on efficient matrix factorization: the LU decomposition, which factors a matrix A into lower (L) and upper (U) triangular components, stands out as the cornerstone technique. Some of the classical proven approaches usually utilized are the Doolittle [8] and Crout [7] algorithms, but when restricted to only symmetric positive definite matrices, the Cholesky method [9] also stands out. While these algorithms share a fundamental mathematical goal, they differ significantly in their inner-loop structures and memory access patterns.

Early optimization efforts focused on numerical stability, as computational advances quickly required the mitigation of floating-point errors through advanced pivoting strategies [10,11]. However, as hardware evolved, the focus shifted toward computational efficiency, leading to practical implementations in standard high-performance libraries such as LAPACK [12] and ScaLAPACK [13], which introduced block-based algorithms to maximize cache locality and overall performance. Recent work has revisited LU factorization through block-elimination schemes with additive modifications that reduce synchronization and communication while maintaining acceptable numerical stability [14]. However, these methods do not address the impact of hybrid CPU architectures on energy consumption, which is the focus of this study.

In the context of modern shared-memory parallelization, the structure of the algorithm plays a decisive role in determining achievable performance. In particular, the Crout algorithm is frequently favored in experimental studies [15] because its dependency structure is well suited to blocked, task-based parallel execution. Furthermore, implementation details

within specific language ecosystems have been shown to significantly impact performance. In Java, libraries such as EJML [16] and Parallel Colt [17] demonstrate that factors including array layout (row-major vs. column-major), primitive handling, and Just-In-Time (JIT) compilation optimizations can be as influential as the algorithm itself [18]. Accordingly, EJML and Parallel Colt provide natural baselines for evaluating new Java-based LU implementations. Their design targets homogeneous multicore systems and therefore they do not capture the effects of heterogeneous core architectures or operating system behavior that motivate our study. This work fills this gap by analyzing LU implementations on hybrid CPUs, with explicit attention to operating system effects and energy efficiency.

2.2. Parallel Execution Models and Energy Consumption

To exploit the capabilities of modern multicore processors, parallel programming models have evolved from rigid, explicitly managed thread-based approaches toward more dynamic and data-driven paradigms. Task-based parallelism, which decomposes computation into smaller, fine-grained tasks, has emerged as a dominant model for expressing concurrency on increasingly complex hardware [19]. By delegating scheduling decisions to a runtime system, task-based models can more effectively adapt to irregular workloads and heterogeneous multicore architectures, improving resource utilization and reducing idle time across processing units [20].

Beyond performance scalability, these execution models have important implications for energy consumption. Runtime systems that manage task scheduling, placement, and synchronization directly influence how cores are utilized, how frequently they transition between power states, and how memory hierarchies are stressed [21,22]. For example, aggressive parallelism may maximize throughput but lead to higher power draw due to increased core activity and higher switching rates, whereas more conservative scheduling strategies can reduce energy usage at the expense of longer execution times [23]. As a result, the efficiency of a parallel execution model must be evaluated not only by how quickly it completes workload but also by how effectively it balances performance with energy costs [24]. In addition, recent work in heterogeneous systems has shown that energy efficiency is strongly influenced by runtime behavior and operating system-level scheduling decisions, in addition to the structure of the numerical kernel itself [25]. These results highlight that the interaction of software runtime, hardware heterogeneity, and power management plays a critical role in determining energy–performance trade-offs.

Moreover, as the emphasis on green software continues to grow in recent years [26], the need to consider energy efficiency as a primary concern in software design has been further reinforced. This perspective highlights that improvements in execution time do not necessarily translate into proportional reductions in energy consumption, as aggressive optimization strategies may increase power draw and offset runtime gains [5,27]. As computing platforms now span from large-scale data centers to resource-constrained personal and edge devices, energy consumption has become a limiting factor not only in terms of operational cost but also with respect to environmental sustainability [28,29]. Consequently, evaluating software systems solely through performance-centric metrics can lead to misleading conclusions, reinforcing the need for energy-aware evaluation methodologies that jointly consider execution time and power usage.

However, these effects have not yet been systematically quantified for parallel LU decomposition. This work addresses this gap by evaluating Java-based LU implementations across heterogeneous and homogeneous architectures under different operating systems.

2.3. System-Level Constraints: Heterogeneity and Operating System Policy

Modern processor design has shifted away from homogeneous multicore systems toward hybrid architectures that integrate P-cores and E-cores within the same chip. While this approach improves flexibility and energy proportionality, it challenges classical performance models that assume identical execution capabilities across threads, making the behavior of parallel numerical algorithms dependent not only on algorithmic complexity or thread count but also on architectural asymmetry. Empirical studies have shown that P-cores dominate floating-point-intensive workloads, whereas E-cores can offer improved energy efficiency under constrained power budgets [30]. Recent evaluations of hybrid processors such as Intel Alder Lake and ARM big.LITTLE platforms report that operating system-level thread placement and hardware feedback mechanisms (e.g., Thread Director) significantly affect both performance and energy efficiency. In several cases, default operating system behavior leads to suboptimal placement of compute-intensive threads, resulting in large performance variability and inefficient energy usage [31]. This imbalance creates a multidimensional performance–energy space in which faster execution does not necessarily imply lower energy consumption, underscoring the need to evaluate both metrics jointly rather than in isolation [32].

Beyond hardware design, the operating system acts as a critical mediator between applications and heterogeneous resources. Thread scheduling, core selection, and power management policies differ substantially across operating systems and even across kernel versions. Prior studies report mixed results when comparing Windows and Linux schedulers for compute-bound workloads, with performance advantages depending on the specific application characteristics and system configuration [33,34]. These variations arise from differences in scheduler heuristics, affinity handling, and default energy policies, which can significantly alter how workloads are mapped onto heterogeneous cores. Studies of heterogeneous systems further indicate that operating system-level resource management and heterogeneity-aware mechanisms can reduce total system energy consumption for CPU-bound workloads, whereas naive load balancing may negate the potential benefits of hybrid architectures [25].

Taken together, these observations suggest that evaluating parallel LU decomposition purely at the algorithmic level is insufficient in modern systems. Performance and energy behavior emerge from the interaction of the chosen LU variant, the underlying processor topology, and the operating system scheduler. While much of the existing literature focuses on only one dimension at a time (algorithm design, operating system behavior, or hardware architecture), this work adopts a holistic perspective by analyzing parallel LU implementations across multiple architectures and operating systems, aiming to clarify how algorithm design choices interact with contemporary hardware and operating system-level resource management [35]. This combined evaluation enables a direct comparison between algorithmic design choices, library-level implementations, and operating system-level effects, addressing gaps left by prior single-layer analyses.

3. Methodology

3.1. LU Decomposition

The LU decomposition is a matrix decomposition method that factors a given matrix A as a product of a lower triangular matrix L and an upper triangular matrix U . This technique is highly used by computers nowadays when solving systems of linear equations efficiently, as this makes them solvable directly by only the use of forward and backward substitutions. Once the LU factorization is computed, multiple right-hand sides can be solved efficiently, in contrast to applying Gaussian elimination repeatedly, which has an arithmetic complexity of $\mathcal{O}(n^3)$ per system.

Several algorithms to compute the LU factorization exist, but for this work, we will be focusing on two main algorithms—Crout and Doolittle—for LU decomposition without permutation. No Cholesky factorization is implemented or evaluated in this study, as the experimental analysis does not restrict input matrices to be symmetric positive definite. For this technique to be feasible, the leading minors of the matrix must satisfy the following condition:

$$\det(A_k) \neq 0 \quad \text{for } k = 1, 2, \dots, n$$

Under this supposition, the factorization to be studied is

$$A = L \times U$$

where A is the initial matrix, and L and U are the lower and upper triangular matrices, respectively. To satisfy this condition in practice, input matrices are generated to be diagonally dominant, which guarantees that all leading principal minors are non-zero and allows LU decomposition without pivoting to be safely applied.

The sequential execution of these algorithms follows the logic presented in Algorithms 1 and 2.

Algorithm 1 Sequential Crout decomposition.

```

1  procedure Crout( $A, L, U, n$ )
2  for  $i = 0$  to  $n - 1$  do  $U_{ii} \leftarrow 1$ 
3  for  $j = 0$  to  $n - 1$  do
4    for  $i = j$  to  $n - 1$  do
5       $sum \leftarrow \sum_{k=0}^{j-1} L_{ik}U_{kj}$ 
6       $L_{ij} \leftarrow A_{ij} - sum$ 
7    end for
8    for  $i = j$  to  $n - 1$  do
9       $sum \leftarrow \sum_{k=0}^{j-1} L_{jk}U_{ki}$ 
10     if  $L_{jj} = 0$  then exit
11      $U_{ji} \leftarrow (A_{ji} - sum) / L_{jj}$ 
12   end for
13 end for
14 end procedure

```

Algorithm 2 Sequential Doolittle decomposition.

```

1  procedure Doolittle( $A, L, U, n$ )
2  for  $i = 0$  to  $n - 1$  do  $L_{ii} \leftarrow 1$ 
3  for  $i = 0$  to  $n - 1$  do
4    for  $j = i$  to  $n - 1$  do
5       $sum \leftarrow \sum_{k=0}^{i-1} L_{ik}U_{kj}$ 
6       $U_{ij} \leftarrow A_{ij} - sum$ 
7    end for
8    for  $j = i$  to  $n - 1$  do
9       $sum \leftarrow \sum_{k=0}^{i-1} L_{jk}U_{ki}$ 
10     if  $U_{ii} = 0$  then exit
11      $L_{ji} \leftarrow (A_{ji} - sum) / U_{ii}$ 
12   end for
13 end for
14 end procedure

```

Custom implementations for the parallel versions of these two algorithms have been developed in Java by taking advantage of its parallelization capabilities. The specific details

regarding task partitioning, thread management, and synchronization mechanisms for these parallel implementations are further elaborated in Section 3.3.

An exception-handling module was also added to prevent the constructed benchmark from terminating abruptly. To verify the correctness of these procedures, various matrices were given as inputs, and outputs were validated without inspecting internal behavior, following a black-box testing approach. Correctness is quantified by computing the relative Frobenius norm of the reconstruction error, a matrix norm commonly used in linear algebra when computing the distance between two matrices. A decomposition is considered valid if $\|A - LU\|_F / \|A\|_F$ remains below a fixed numerical tolerance, indicating that $L \times U$ is sufficiently close to the original matrix. The Frobenius norm is computed as

$$\|A\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^m |a_{ij}|^2}$$

for any matrix A of size $n \times m$.

3.2. Runtime and Consumption Measurements

Computational performance was assessed using speedup as the primary metric, defined as the ratio between the execution time of the best sequential implementation and that of the parallel version executed with multiple threads:

$$Speedup = \frac{t_{sequential}}{t_{parallel}}$$

The sequential execution corresponds to the best-performing single-threaded implementation of the same algorithm and matrix size.

Also, due to the intrinsic uncertainty of time measurements on electronic devices, executions were repeated ten times for each architecture–algorithm–size polynomial, computing the mean execution time value that would be later used to obtain the representative speedup value for that specific configuration. Moreover, to reduce the impact of Java JIT compilation, several warm-up executions were performed prior to measurement and discarded from the analysis.

Figure 1 illustrates the complete measurement workflow. For each combination of architecture (described in Section 4), algorithm (Crout or Doolittle), matrix size (from 3000 to 10,000 in increments of 1000), and thread count, 10 independent execution times were recorded. Each execution consisted of an LU decomposition together with its corresponding black-box test. Finally, the mean runtime and the resulting speedup were computed for each configuration.

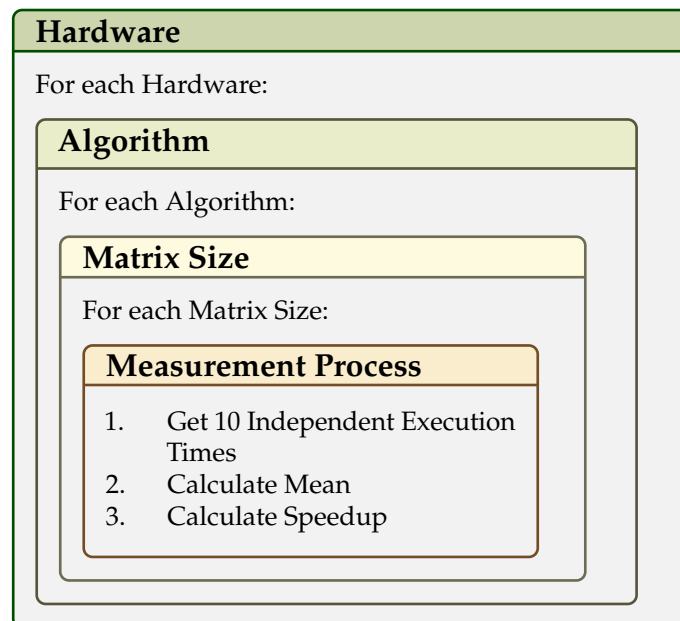


Figure 1. Nested methodology for runtime measurement and speedup evaluation.

In addition to performance, power consumption was also measured for every experimental setup in order to provide a comprehensive assessment of the energy efficiency of the Java implementations. These measurements enable an analysis of how different algorithmic choices interact with underlying hardware characteristics, highlighting trade-offs between execution speed and energy usage. By examining power behavior across varying architectures, operating systems, and problem sizes, this study offers insights into the balance between computational efficiency and energy cost.

Energy measurements reflect whole-system consumption, including processor, memory, and other active system components. This choice enables a realistic evaluation of the energy impact of parallel LU decomposition in practical execution environments.

3.3. Parallel Implementation Details

The parallel versions of the Crout and Doolittle algorithms were implemented from scratch using the Java Concurrency API. The implementation is designed around three main aspects: thread management, workload distribution, and synchronization.

Instead of creating and destroying threads for each computational task, which would introduce significant overhead, the implementation employs an `ExecutorService` with a `FixedThreadPool`. The number of threads in the pool is configured according to the experimental setup, ranging from sequential execution up to 272 threads in many-core environments such as the Intel Xeon Phi. This approach keeps thread management overhead low and avoids repeated thread creation during the factorization.

The LU decomposition exhibits inherent data dependencies: certain entries of the L and U factors must be computed before others. To parallelize the computation while respecting these dependencies, we adopt a cyclic (interleaved) distribution strategy:

- The outer-loop index j remains sequential, since it determines the progression of the factorization and enforces the required dependencies.
- For each fixed j , the inner-loop iterations that compute the elements of the current column of L or the current row of U are partitioned across a thread pool.
- Each thread is assigned a subset of target indices starting at $i = j + \text{thread_id}$, where i denotes the computed index and thread_id is the thread identifier. The thread then advances in strides of T (i.e., $i \leftarrow i + T$), where T is the total number of threads.

To guarantee numerical correctness, the implementation employs `CountDownLatch` as a synchronization barrier. At each decomposition step, a latch is initialized with count T , and one task per `thread_id` is submitted to the thread pool. Each task computes its assigned portion of the matrix and invokes `latch.countDown()` upon completion, while the coordinating thread waits via `latch.await()` until all tasks have finished before proceeding to the next step. The resulting parallel execution scheme is described in Algorithm 3.

Algorithm 3 Parallel LU step logic.

```

1  Create latch with count  $T$ 
2  for each thread_id  $\in \{0, \dots, T - 1\}$  do
3    P.execute({
4      for  $i = j + \text{thread\_id}$  to  $n - 1$  step  $T$  do
5        Compute element ( $L_{ij}$  or  $U_{ji}$ ) using sum reduction
6      end for
7      latch.countDown()
8    })
9  end for
10 latch.await() // Barrier synchronization

```

This mechanism ensures that all results produced in the current step are completed and visible before they are used as inputs for subsequent calculations.

4. Experimental Setup

The experimental setup was established to assess both computational performance and energy consumption for sequential and parallel LU decomposition algorithms, specifically the Crout and Doolittle methods, implemented in Java. This study was conducted across four hardware platforms chosen to represent a diverse set of modern architectural designs:

- *Core i9*: An Intel Core i9-12900HK hybrid processor with 6 P-cores and 8 E-cores (14 cores and 20 threads), featuring dynamic frequency scaling up to 5.0 GHz. Evaluated on both Windows 11 and Ubuntu 24.04 LTS.
- *Ryzen 7*: An AMD Ryzen 7 7800X3D homogeneous desktop processor with 8 Zen 4 cores and 16 threads, operating at boost frequencies of up to 5.0 GHz. Evaluated on both Windows 11 and Ubuntu 24.04 LTS.
- *Xeon Phi*: An Intel Xeon Phi 7250 many-core processor with 68 homogeneous cores and 272 hardware threads, optimized for massively parallel workloads at 1.4 GHz. Evaluated on Ubuntu 22.04 LTS.
- *Mac Mini*: Apple Mac Mini with the Apple M4 system on chip, combining 4 P-cores and 6 E-cores (10 cores total) in a unified memory architecture. Evaluated on macOS Sequoia 15.1.

All parallel LU decomposition algorithms were implemented and executed using OpenJDK 21. The Java Virtual Machine (JVM) was launched with default parameters, specifically without the use of incubator modules or explicit SIMD flags (e.g., `--add-modules jdk.incubator.vector`). Consequently, any vectorization achieved was solely the result of the HotSpot JIT compiler's auto-vectorization capabilities, rather than explicit use of the Java Vector API.

The hardware configurations included two heterogeneous architectures that combine P- and E-cores (Core i9 and Mac Mini), as well as two homogeneous architectures (Ryzen 7 and Xeon Phi). All tests were performed using square matrices with dimensions ranging from $n = 3000$ (3000×3000) to $n = 10,000$ ($10,000 \times 10,000$).

To evaluate energy efficiency, power consumption was measured externally in order to capture the total system power draw. An industrial-grade *NETIO PowerCable REST 101F* smart socket was used to record power usage in real time throughout program execution. To synchronize power measurements with algorithm execution, timestamps were recorded both at the start and end of each run and aligned with the corresponding power samples, ensuring that only the energy consumed during the effective execution interval was considered. The total energy consumption E was computed as

$$E = \sum_{i=1}^n P_i dt$$

where n denotes the total number of collected samples, dt is the sampling interval, and P_i represents the instantaneous power at sample i . In addition, the average power consumption $\bar{P}$ was obtained as

$$\bar{P} = \frac{E}{\Delta t}$$

where Δt denotes the total execution time of the algorithm for each specific configuration.

5. Results

This section exhibits the results of the execution time and energy consumption measurements for the architectures studied. The analysis presented focuses on matrices of size $n = 10,000$. This choice was made to reduce visual complexity and to facilitate consistent comparisons across hardware platforms, operating systems, and algorithmic variants while still capturing representative performance and energy behavior. Complete results for all evaluated configurations are provided in Appendices A and B.

5.1. Impact of Parallel LU Matrix Decomposition Algorithms on Execution Time

Figure 2 presents the performance results for the Crout and Doolittle implementations on the Mac Mini platform. The left panel shows the achieved speedup (Figure 2a), while the right panel reports the execution time measured in seconds (Figure 2b). Together, these visualizations allow for a direct comparison of the scalability and absolute runtime behavior of both algorithms under the same hardware configuration.

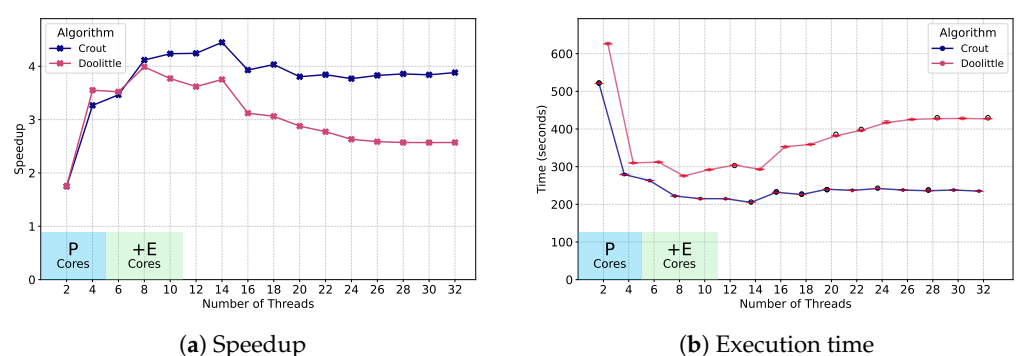


Figure 2. Crout and Doolittle implementations on the Mac Mini. System configuration: 4 P-cores (represented by the blue-shaded area) and 6 E-cores (green-shaded area), for a total of 10 threads.

It can be observed that the speedup curve of the Crout algorithm continues to increase until reaching 14 threads. However, this growth is not uniform across all thread counts, with the most significant performance gains occurring between 2 and 4 threads and between 6 and 8 threads. In contrast, Doolittle exhibits different behavior: its speedup increases only up to 8 threads, with a slight drop observed between 4 and 6 threads. Beyond this point, the speedup decreases noticeably as the number of threads increases. These results indicate

that P-cores contribute more effectively to scalability, whereas the addition of E-cores yields limited benefits in terms of runtime performance.

Figure 3 describes the performance results for the Crout and Doolittle implementations on the Core i9 platform, evaluated under both Linux and Windows operating systems. Focusing on the Linux results (see Figure 3a,b), both algorithms exhibit an initial increase in performance as the number of threads grows, up to the point where the thread count matches the number of available P-cores. Beyond this point, a sharp performance drop is observed for both methods, although with different magnitudes. Crout is more severely affected, with its speedup decreasing to values close to those obtained using only two threads. In contrast, Doolittle experiences a more moderate reduction, losing approximately half of the performance drop observed for Crout. After this degradation, both algorithms gradually recover performance as additional threads are employed, eventually converging to similar speedup values at 20 threads, corresponding to the maximum number of logical threads supported by the processor. Beyond this point, performance continues to increase, albeit at a much slower rate.

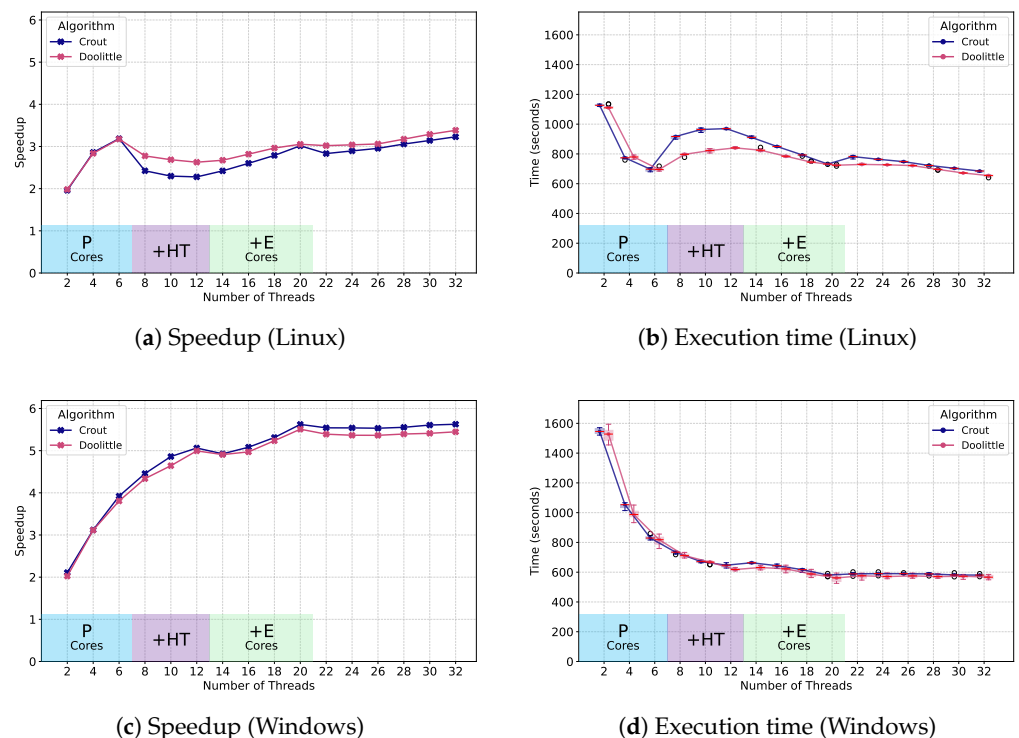


Figure 3. Crout and Doolittle implementations on the Core i9. System configuration: 6 P-cores (represented by the blue-shaded area) with Hyper-Threading (purple-shaded area) and 8 E-cores (green-shaded area), for a total of 20 threads.

Regarding the Windows results (see Figure 3c,d), both algorithms exhibit a similar scalability trend. However, Crout consistently achieves higher performance across all thread counts. In this case, speedup increases up to 12 threads, corresponding to the number of performance cores with Hyper-Threading enabled. A slight performance drop is then observed, followed by a rapid recovery, after which both algorithms continue to improve until reaching 20 threads, the maximum number of logical threads supported by the processor. Beyond this point, the speedup values for both methods remain essentially unchanged. Overall, these results indicate that scalability on the Core i9 is once more largely driven by P-cores, while E-cores provide limited additional runtime performance benefits.

Figure 4 reports the performance results obtained for the Crout and Doolittle algorithms on the AMD Ryzen 7 processor on Linux and Windows. On Linux (see Figure 4a,b), Crout and Doolittle exhibit nearly identical performance behavior. Speedup increases steadily up to 8 threads, corresponding to the number of physical cores, after which a performance drop is observed. When using 16 threads, enabled through simultaneous multi-threading, partial recovery occurs, although the resulting performance gain remains modest. Beyond this point, speedup decreases again and subsequently recovers slightly but does not reach the level achieved when all logical threads are first utilized.

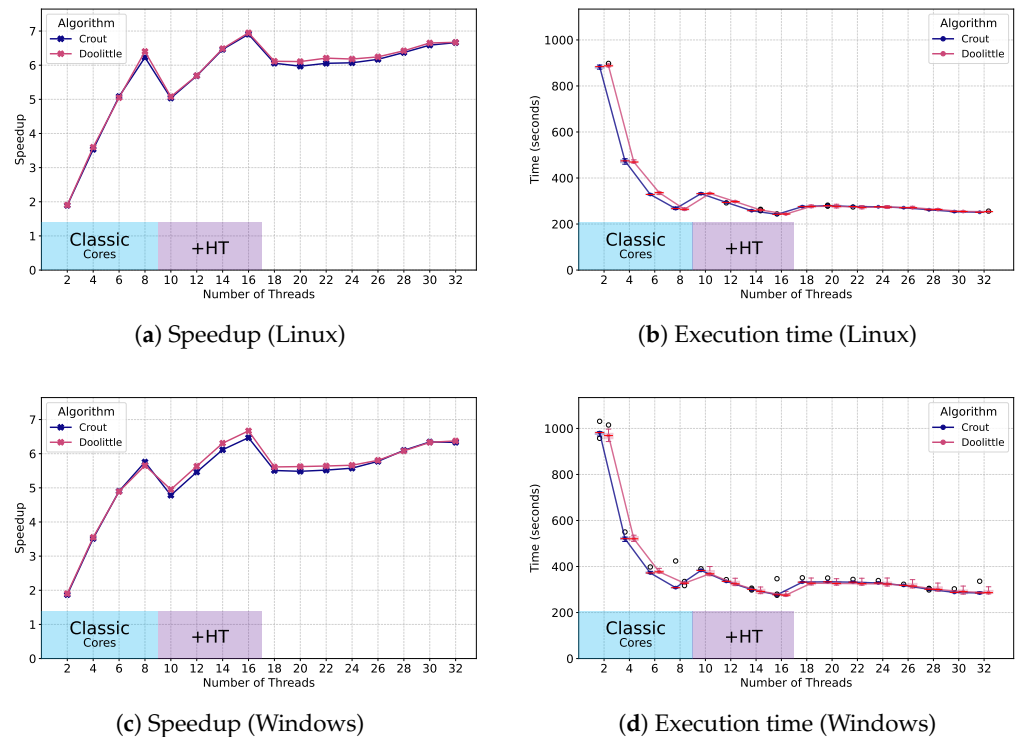


Figure 4. Crout and Doolittle implementations on the Ryzen 7. System configuration: 8 cores (represented by the blue-shaded area) with simultaneous multi-threading (purple-shaded area), for a total of 16 threads.

On Windows (see Figure 4c,d), the two algorithms exhibit different performance trends. Both Crout and Doolittle scale up to 8 threads, corresponding to the number of physical cores, with Crout achieving higher speedup in this region. When the thread count increases to 16, enabled through simultaneous multi-threading, Doolittle outperforms Crout. Beyond this point, both curves decline following a similar pattern and subsequently exhibit a slight recovery as additional threads are employed. These results suggest that scalability on the Ryzen platform is largely constrained by the number of physical cores, with limited gains from simultaneous multi-threading.

Finally, Figure 5 presents the speedup and execution time results for the Intel Xeon Phi platform. In this case, Doolittle outperforms Crout up to 112 threads. Beyond this threshold, both algorithms exhibit a gradual increase in performance. However, after approximately 208 threads, Doolittle's speedup begins to decline, whereas Crout maintains stable performance up to around 224 threads, after which it also decreases. At higher thread counts, Crout ultimately achieves higher speedup values than Doolittle.

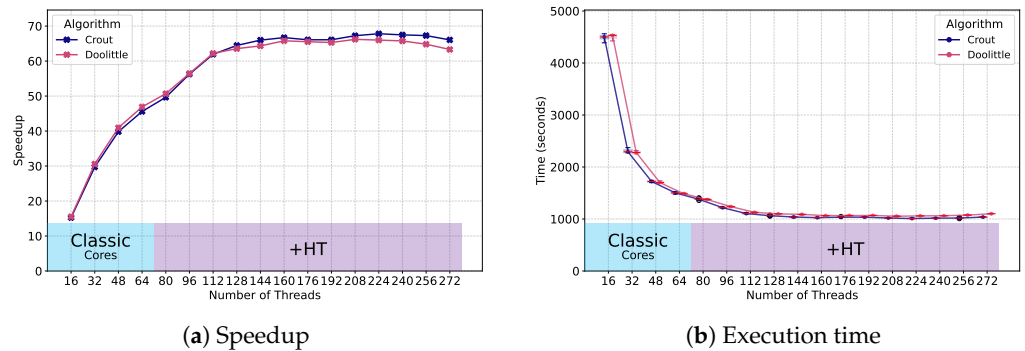


Figure 5. Crout and Doolittle implementations on the Xeon Phi. System configuration: 68 cores (represented by the blue-shaded area) with 4 hardware threads per core (purple-shaded area), for a total of 272 threads.

Taken together, these results highlight how architectural design plays a decisive role in the scalability of parallel LU decomposition. While homogeneous architectures such as the Ryzen 7 and Xeon Phi exhibit performance trends closely tied to core count and threading capabilities, heterogeneous systems introduce more complex, non-linear behaviors driven by the interaction between different core types. In particular, P-cores consistently dominate scalability, whereas additional hardware threads or E-cores provide diminishing and workload-dependent benefits. These observations reinforce the importance of jointly considering algorithmic characteristics and architectural features when evaluating parallel performance.

5.2. Impact of Parallel LU Matrix Decomposition Algorithms on Energy Consumption

Figure 6 summarizes the total energy consumption of the Crout and Doolittle LU decomposition implementations across all evaluated platforms and operating systems. The figure reports whole-system energy usage, measured in kilojoules (kJ), for increasing levels of parallelism, allowing for a direct comparison of the energy efficiency trends of both algorithms under different architectural and software configurations.

Regarding energy consumption on the Mac Mini, Figure 6a shows a clear downward trend in total energy usage as the degree of parallelism increases. For the Crout algorithm, the energy required to complete the computation decreases sharply until reaching between 10 and 14 threads, which corresponds to the point of optimal energy efficiency. Energy consumption is reduced from 13,643 J in the sequential execution to a minimum of 8275 J when using 10 threads. Beyond 14 threads, energy consumption increases slightly and then stabilizes, exhibiting only minor fluctuations. Doolittle follows a nearly identical trend, also reaching its maximum energy efficiency at 10 threads. In both cases, total energy savings of approximately 40% are achieved when moving from sequential to the most energy-efficient parallel configuration.

On the Xeon Phi, energy consumption exhibits a specialized energy profile, characterized by a pronounced exponential reduction in energy consumption as the workload is distributed across its many-core architecture (see Figure 6b). Energy consumption decreases sharply from a sequential value exceeding 12,000 kJ to approximately 800 kJ with only 16 threads, and eventually reaches an optimal efficiency plateau of about 200 kJ between 144 and 208 threads. In this stable regime, the Crout and Doolittle algorithms exhibit nearly identical energy behavior.

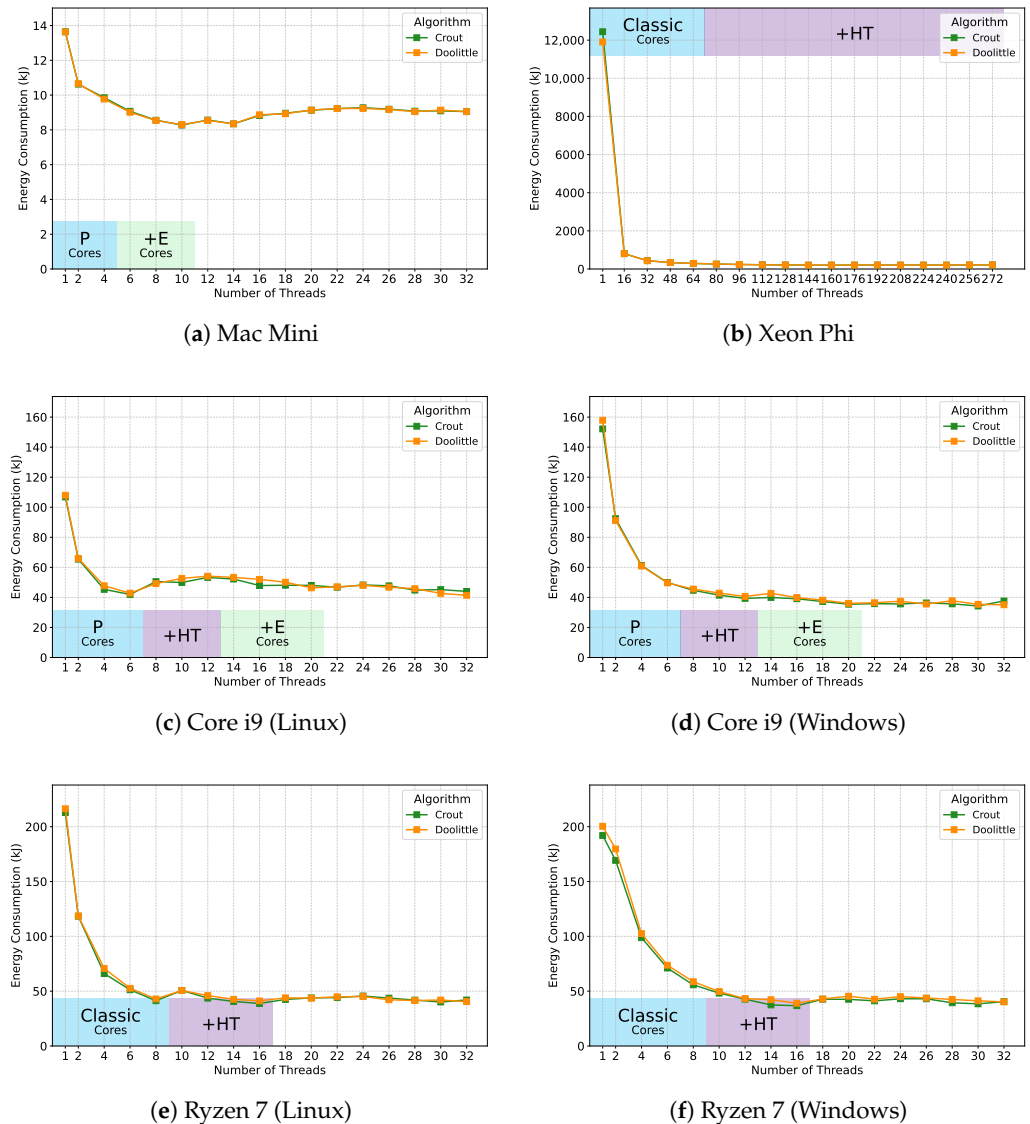


Figure 6. Total energy consumption of the Crout and Doolittle implementations across all evaluated platforms and operating systems. The blue-shaded area represents P-cores or classic cores, as the case may be; purple indicates where Hyper-Threading has become active, while green means that the E-cores have also activated.

Measurements on the Core i9 architecture under Linux reveal a drastic reduction in energy consumption between 1 and 6 threads for both algorithms (see Figure 6c). However, a notable energy spike is observed between 8 and 14 threads, during which the consumed energy increases substantially (Crout rises from approximately 42 kJ with 6 threads to about 53 kJ with 12 threads). Energy efficiency begins to recover only as the thread count approaches 32, where the reduction in execution time is sufficient to offset the higher power draw of the full processor package.

On Windows, the Core i9 exhibits a more gradual and consistent reduction in energy consumption than on Linux (see Figure 6d). Both algorithms show a steady decline in energy usage, reaching an efficiency point at 20 threads. In contrast to Linux, the consumption spike is much less pronounced, although total energy values are generally higher on Windows (reaching 152 kJ for sequential Crout). The Crout algorithm consistently outperforms Doolittle in terms of energy efficiency on this operating system, requiring

fewer joules across nearly all thread counts. Beyond 20 threads, energy consumption remains largely stable.

On Linux, the Ryzen 7 exhibits a pronounced efficiency peak (see Figure 6e). For both algorithms, energy consumption reaches its minimum at 8 threads. In this configuration, energy usage drops from 212 kJ to approximately 41 kJ. When simultaneous multi-threading is enabled (between 10 and 16 threads), a slight increase in energy consumption is observed. Beyond 16 threads, consumption stabilizes, indicating that the processor achieves optimal energy efficiency when using one thread per physical core.

On Windows, the Ryzen 7 exhibits a different behavior (see Figure 6f). Although energy consumption decreases substantially relative to sequential execution, the reduction is more gradual and reaches its minimum later, between 14 and 16 threads. In addition, Windows shows greater variability in energy consumption at low thread counts, particularly between 2 and 4 threads. Once the limit of 16 logical threads is exceeded, energy consumption increases slightly and then remains largely stable for both algorithms.

Overall, the results indicate that the energy efficiency of parallel LU decomposition depends strongly on both the hardware architecture and the operating system. Although parallel execution consistently reduces energy consumption compared with sequential runs, each platform exhibits a distinct optimal thread count beyond which efficiency gains diminish. Many-core systems benefit from high levels of parallelism, while conventional multicore processors achieve maximum efficiency closer to their number of physical cores. Across all configurations, Crout and Doolittle show similar energy trends, with Crout being slightly more energy-efficient in several cases.

Beyond these general trends, the practical implication is that default parallel execution strategies can be significantly suboptimal on hybrid processors. For example, on the Core i9 platform on Linux, running with all 20 logical threads consumes about 48 kJ, whereas selecting the empirically optimal configuration of 6 threads reduces energy consumption to 42 kJ (a 12.5% improvement) while also avoiding the performance degradation observed in the 8–14 thread range. This illustrates how hybrid architectures exhibit efficiency traps that are not visible without detailed energy–performance analysis.

6. Discussion of the Results

This discussion is structured around the research questions defined in Section 1. **RQ1** investigates how heterogeneous architectures influence the runtime of the considered parallel LU decomposition algorithms. **RQ2** examines how these heterogeneous architectures affect the energy consumption of the same parallel LU decomposition algorithms. **RQ3** explores the relationship between runtime and energy consumption for the considered parallel LU decomposition when executed on heterogeneous architectures.

RQ1. How do heterogeneous architectures affect the runtime of the considered parallel LU decomposition algorithms?

Heterogeneous architectures influence the runtime scalability of parallel LU decomposition by introducing performance behaviors that depend on the relative capabilities of different core types and on how the system schedules work across them. Compared with homogeneous platforms, hybrid designs tend to exhibit less uniform scaling, with changes in the marginal benefit of additional threads as execution transitions across core classes or hardware threading modes.

Across the evaluated systems, the largest runtime improvements are associated with effectively exploiting the highest performance resources (e.g., P-cores). Beyond that point, additional parallelism provided by E-cores or extra hardware threads can still improve runtime but often with reduced and more irregular gains, and in some configurations, it

can even lead to temporary performance regression. This effect is most clearly observed on Mac Mini (see Table A1), where the optimal configuration is bounded by the number of physical cores (P-cores + E-cores). While using slightly more threads than cores can still be beneficial, further increasing the thread count mainly leads to contention for shared caches and memory bandwidth, yielding little or no performance improvement while increasing energy consumption. Therefore, over-threading should be avoided on these architectures when efficiency is a primary concern.

Operating system behavior further modulates these effects. The contrast between the Core i9 results on Linux and Windows illustrates that scheduling and frequency scaling policies can amplify or mitigate the performance impact of heterogeneity, even when the hardware and algorithms remain unchanged. For Windows, the utilization of Intel's Thread Director better reflects the potential of these new hybrid processors, as it provides far better scheduling policies when using both P-cores and E-cores simultaneously. On Linux, in our experimental setup, when E-cores become active, per-core frequency measurements show that P-cores no longer sustain their turbo frequencies, which contributes to the performance collapse at high thread counts (see the speedup values in Table A2). This behavior arises from the lack of hybrid-aware scheduling support.

In contrast, homogeneous architectures show more predictable runtime trends, with scalability primarily bounded by physical core count and limited benefits from simultaneous multi-threading at higher thread counts. Overall, the results suggest that architectural heterogeneity complicates runtime scalability by making performance gains increasingly dependent on how parallelism is mapped to the available compute resources rather than on maximum parallelism alone.

Heterogeneous architectures lead to less uniform runtime scaling: P-cores drive most of the performance gains, while E-cores and additional hardware threads often provide smaller and more workload- and operating system-dependent benefits.

RQ2. How do heterogeneous architectures affect the energy consumption of the considered parallel LU decomposition algorithms?

Heterogeneous architectures have a pronounced impact on the energy consumption of parallel LU decomposition, primarily because energy efficiency depends on how workloads are distributed across different core types and on operating system scheduling and frequency scaling. While parallel execution reduces total energy consumption relative to sequential runs in the evaluated systems, the energy benefits of increasing parallelism are not uniform across architectures.

In hybrid systems, the most energy-efficient configuration is platform- and operating system-dependent. Increasing parallelism can improve efficiency up to a characteristic thread range, but engaging additional E-cores or hardware threads may lead to diminishing returns and, in some cases, temporary energy penalties (as observed on the Core i9 on Linux). In other cases, the energy minimum can occur when the full set of cores is effectively utilized (e.g., the Mac Mini around 10 threads).

In contrast, homogeneous architectures exhibit more stable and predictable energy efficiency trends. Their optimal configurations tend to align with the use of physical cores, while additional logical threads provide limited energy benefits and may increase overall power consumption.

At low thread counts, energy consumption is strongly influenced by the combined effect of hardware and operating system. For instance, on the Ryzen 7 platform (see Table A7), the sequential configuration often shows lower energy on Windows than on Linux (particularly for Crout), whereas on the Intel i9 (see Table A6), the opposite trend

is observed. These differences are not attributable to algorithmic behavior but are consistent with power management effects that vary across different hardware–operating system combinations.

Taken together, these results indicate that heterogeneity increases the sensitivity of energy consumption to thread count and system-level decisions. Although parallelism generally improves energy efficiency, the most energy-efficient configuration is architecture-dependent and cannot be assumed to correspond to maximum parallelism.

Heterogeneous architectures make energy efficiency highly sensitive to how parallelism is mapped onto different core types and managed by the operating system, leading to architecture-dependent optimal configurations that do not necessarily coincide with maximum parallelism.

RQ3. What is the relationship between runtime and energy consumption for the considered parallel LU decomposition on heterogeneous architectures?

The relationship between runtime and energy consumption on heterogeneous architectures is quantified using the Spearman rank correlation coefficient, computed for each hardware–algorithm–operating system combination and reported in Appendix C at a 95% confidence level. This non-parametric measure is selected for its robustness to non-linear relationships and outliers, which are common in multicore performance data. Correlation values close to one indicate that reductions in execution time are strongly associated with decreases in total energy consumption, whereas lower or negative values reveal the presence of performance–energy trade-offs. The statistical significance of each correlation is evaluated through the corresponding Spearman p-value, with low values indicating a reliable association and higher values suggesting weaker or inconclusive relationships.

The correlation analysis reveals that the coupling between runtime and energy consumption generally strengthens as matrix size increases. For most configurations, larger problem sizes yield correlation coefficients closer to one, indicating a more consistent alignment between performance improvements and energy savings. However, this trend is not uniform across all platforms. On the Mac Mini and the Ryzen 7, smaller matrices exhibit weaker and often statistically insignificant correlations, reflecting unstable or less predictable time–energy relationships at low computational intensity. This behavior reflects an energy efficiency degradation caused by the attempt to parallelize small workloads, where JVM and operating system overhead (e.g., JIT compilation, garbage collection, and thread management) dominate the useful computation. In contrast, for sufficiently large matrices the Intel platforms exhibit high and statistically significant correlations, indicating a strong coupling between execution time and energy consumption.

Overall, these results show that while runtime is often a good proxy for energy efficiency, this relationship depends on both workload size and architectural characteristics. The commonly used “race-to-sleep” strategy is, therefore, effective for large-scale computations, but for small problems, the parallelization overhead can outweigh the benefits, leading to reduced energy efficiency rather than savings. On heterogeneous systems in particular, the presence of multiple core types and system-level scheduling effects can further weaken the correspondence between performance and energy.

This study demonstrates that heterogeneity-aware configuration based on empirical energy–performance analysis provides clear advantages over fixed, hardware-agnostic execution choices. Although Doolittle is often used as a standard reference, our results indicate that Crout’s task-dependency structure yields more favorable energy–delay trade-offs on Windows-managed hybrid systems. By comparing algorithms, thread counts,

and platforms in a unified framework, the proposed analysis enables the identification of energy-efficient configurations that would not be revealed by performance-only evaluation.

The correlation analysis confirms that faster execution generally leads to lower energy consumption, but the strength and reliability of this relationship depend on architecture and problem size, with heterogeneous systems exhibiting workload-dependent time–energy trade-offs.

7. Threats to Validity

Construct validity. Execution time is measured using wall-clock time, which may be affected by operating system noise and background activities. Although multiple executions are averaged and warm-up runs are discarded to mitigate JVM effects, some variability may still remain. In addition, energy consumption is measured at the whole-system level, which includes components other than the CPU and memory subsystem.

Internal validity. Differences in thread scheduling, frequency scaling, and power management policies across operating systems may influence the observed performance and energy behavior. While experiments are conducted under controlled and repeatable conditions, it is not possible to fully eliminate all sources of runtime interference inherent to modern operating systems and heterogeneous processors.

External validity. This study focuses on dense square matrices of specific sizes and on a limited set of processor architectures and operating systems. As a result, the findings may not generalize directly to sparse matrices, other numerical workloads, or future processor designs with different heterogeneity characteristics.

Conclusion validity. Statistical conclusions are based on a finite number of executions per configuration and on non-parametric correlation analysis. Although these methods are chosen for robustness, larger sample sizes or alternative statistical techniques could further strengthen the confidence in the reported relationships.

8. Conclusions and Future Work

This study provides a rigorous experimental assessment of the execution time and energy consumption of parallel LU decomposition algorithms in Java, specifically the Crout and Doolittle variants. The findings demonstrate that the hardware and software configuration yielding the maximum speedup does not necessarily correspond to the one that minimizes energy consumption. Across four representative platforms, including the Apple M4, Intel Core i9, AMD Ryzen 7, and Intel Xeon Phi, we observed that energy-aware optimization strategies are essential, as aggressive parallelism often leads to diminishing returns in energy efficiency.

Architectural heterogeneity plays a decisive role in shaping performance–energy trade-offs. On the Mac Mini, energy consumption is reduced by approximately 40% at an optimal configuration of 10 threads, even though performance continues to scale slightly up to 14 threads. In contrast, the Core i9 architecture under Linux exhibits a clear *negative trade-off zone* between 8 and 14 threads, where the activation of Hyper-Threading and E-cores results in both slower execution times and increased energy consumption. Similarly, the Ryzen 7 processor reaches its minimum energy usage when operating with 8 physical cores, confirming that a one-to-one mapping between threads and physical cores is the most energy-efficient choice for energy-constrained workloads on this platform.

In contrast to consumer-grade hardware, the Xeon Phi demonstrated massive scalability, achieving speedup above 60× and reaching an energy efficiency plateau between 144 and 208 threads. The analysis shows that while low-frequency many-core architectures are highly inefficient for sequential workloads, they become energy-efficient when

operating with two hardware threads per core, effectively mitigating memory latency. However, increasing parallelism to the maximum of four threads per core introduces resource contention, resulting in stagnant performance and increased power consumption.

The comparative analysis indicates that the proposed energy–performance evaluation approach consistently reveals more favorable energy–delay trade-offs than default parallel execution strategies. In hybrid environments, using the maximum available level of parallelism, as is common in numerical libraries, can lead to unnecessary energy overhead. By identifying architecture-specific efficiency optima, our analysis achieves up to 40% energy savings on the Apple M4 and avoids pronounced energy spikes on Intel platforms, highlighting the practical benefits of energy-aware configuration on modern asymmetric hardware.

Future work will focus on expanding this methodology to sparse matrix operations and alternative numerical workloads to determine if these energy-proportionality trends remain consistent. Furthermore, there is a clear need to develop automated thread-management frameworks that leverage the metrics to dynamically adjust thread-usage levels in real time based on the detected hardware topology. Additional research could also involve comparing Java’s performance–energy profile against native languages like C to evaluate the specific energy impact of the JVM’s JIT compilation on heterogeneous multicore processors.

Author Contributions: Conceptualization, J.C.d.l.T. and F.J.R.; methodology, J.C.d.l.T., F.J.R. and J.M.A.-J.; software, F.J.R. and A.V.-G.; validation, J.C.d.l.T., J.M.A.-J., and B.D.; formal analysis, J.C.d.l.T., J.M.A.-J. and B.D.; investigation, F.J.R., J.C.d.l.T., J.M.A.-J., A.V.-G. and B.D.; resources, B.D.; data curation, F.J.R. and A.V.-G.; writing—F.J.R. and A.V.-G.; writing—review and editing, J.C.d.l.T., J.M.A.-J. and B.D.; visualization, J.M.A.-J.; supervision, J.C.d.l.T.; project administration, B.D.; funding acquisition, B.D. All authors have read and agreed to the published version of the manuscript.

Funding: This publication is part of the GRASS (FEDER-UCA-2024-A2-23) and gCODE (PIDI_2024_01307) projects, both funded by European Regional Development Funds (ERDFs) and Consejería de Universidad, Investigación e Innovación de la Junta de Andalucía. J.M. Aragón-Jurado would like to acknowledge the Spanish Ministerio de Ciencia, Innovación y Universidades for the support through grant FPU21/02026. Juan Carlos de la Torre would like to acknowledge the University of Cadiz for its support through the project “Técnicas avanzadas de Inteligencia Artificial aplicadas al Tratamiento de Imágenes en entornos reales” (I+i 2024/03).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data and source code supporting the findings of this study are openly available in the GitHub repository *Energy–Performance Trade-offs of LU Matrix Decomposition in Java Across Heterogeneous Hardware and Operating Systems* at <https://github.com/goal-group-uca/-Applied-Sciences-Energy-Performance-Trade-offs-of-LU-Matrix-Decomposition-in-Java->.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Detailed Speedup Results

This appendix provides the complete set of experimental results regarding computational performance. The tables present the speedup values for both the Crout and Doolittle algorithms across all evaluated hardware platforms (Mac Mini M4, Intel Core i9-12900HK, AMD Ryzen 7 7800X3D, and Intel Xeon Phi 7250) and operating systems. For each matrix size (ranging from $n = 3000$ to $n = 10,000$), the maximum speedup achieved is highlighted in bold to identify the most performant thread configuration.

Table A1. Speedup results on Mac Mini M4. Values grouped by algorithm and operating system. Bold values indicate the maximum speedup achieved for each matrix size, identifying the most performant thread configuration.

Mac Mini M4																
Algorithm: Crout																
macOS	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.46	2.58	2.74	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44	3.44
4000	1.63	3.12	3.24	3.83	3.95	4.22	4.67	4.22	4.22	3.83	4.20	3.83	4.22	4.22	4.22	4.22
5000	1.76	3.30	3.50	4.11	4.29	4.69	4.69	4.19	4.48	4.08	4.11	4.08	4.11	4.15	4.12	4.29
6000	1.73	3.27	3.46	4.10	4.21	4.56	4.72	4.20	4.41	4.02	4.15	3.99	4.12	4.19	4.14	4.20
7000	1.75	3.28	3.44	4.12	4.25	4.59	4.77	4.24	4.49	4.09	4.21	4.01	4.18	4.16	4.13	4.18
8000	1.75	3.28	3.42	4.14	4.24	4.49	4.64	4.16	4.39	4.00	4.14	3.96	4.07	4.10	4.05	4.08
9000	1.75	3.27	3.39	4.12	4.22	4.36	4.54	4.04	4.19	3.89	3.96	3.87	3.98	4.00	3.97	3.99
10000	1.75	3.26	3.46	4.12	4.24	4.24	4.44	3.93	4.02	3.80	3.84	3.77	3.83	3.86	3.84	3.88
Algorithm: Doolittle																
macOS	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.64	2.88	2.88	2.88	2.88	2.88	2.88	2.30	2.30	2.30	2.30	2.30	2.30	2.30	2.30	2.30
4000	1.57	3.25	3.13	3.58	3.85	3.85	3.85	3.13	3.13	2.95	2.95	2.78	2.93	2.95	2.95	2.95
5000	1.68	3.41	3.50	3.51	3.91	3.79	3.91	3.24	3.34	2.98	2.92	2.83	2.67	2.71	2.77	2.77
6000	1.73	3.55	3.61	3.84	4.25	4.28	4.41	3.58	3.74	3.32	3.32	3.14	2.95	2.93	2.99	3.00
7000	1.75	3.62	3.66	3.87	4.17	4.21	4.32	3.55	3.64	3.27	3.23	3.02	2.85	2.80	2.88	2.88
8000	1.75	3.61	3.57	3.83	4.06	4.03	4.17	3.43	3.45	3.17	3.09	2.91	2.77	2.75	2.77	2.76
9000	1.73	3.56	3.46	3.76	3.88	3.74	3.88	3.22	3.19	3.00	2.90	2.72	2.65	2.63	2.62	2.63
10000	1.75	3.55	3.52	3.99	3.77	3.62	3.75	3.12	3.06	2.88	2.77	2.63	2.59	2.57	2.57	2.57

Table A2. Speedup results on Intel Core i9-12900HK. Values grouped by algorithm and operating system. Bold values indicate the maximum speedup achieved for each matrix size, identifying the most performant thread configuration.

Intel i9-12900HK																
Algorithm: Crout																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.85	2.27	2.00	1.39	1.42	1.49	1.58	1.71	1.74	1.83	1.83	1.89	1.96	1.97	1.98	2.04
4000	1.96	2.66	2.47	1.55	1.60	1.69	1.83	1.98	2.08	2.19	2.27	2.35	2.45	2.52	2.59	2.59
5000	2.02	2.78	2.87	1.86	1.92	2.01	2.17	2.35	2.51	2.65	2.72	2.84	2.93	3.03	3.10	3.14
6000	1.98	2.76	2.89	1.74	1.76	1.88	2.03	2.19	2.34	2.50	2.47	2.58	2.69	2.84	2.98	3.02
7000	2.00	2.85	3.04	1.91	1.91	2.00	2.18	2.35	2.53	2.69	2.61	2.71	2.80	2.96	3.13	3.18
8000	1.99	2.82	3.07	2.01	1.95	2.03	2.19	2.37	2.56	2.74	2.62	2.69	2.80	2.93	3.06	3.09
9000	1.98	2.84	3.19	2.22	2.11	2.14	2.28	2.48	2.65	2.86	2.70	2.77	2.86	2.98	3.10	3.18
10000	1.96	2.84	3.21	2.42	2.30	2.28	2.42	2.60	2.78	3.02	2.85	2.89	2.95	3.05	3.14	3.22
Algorithm: Doolittle																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.00	2.90	3.38	3.59	3.84	3.93	4.04	3.80	3.80	3.84	3.82	3.88	4.04	4.15	4.11	4.16
4000	2.12	3.09	3.65	4.03	4.26	4.49	4.27	4.09	4.32	4.48	4.48	4.49	4.57	4.64	4.50	4.43
5000	2.08	3.11	3.82	4.43	4.84	5.18	4.86	4.84	5.11	5.42	5.33	5.38	5.44	5.52	5.45	5.35
6000	2.13	3.13	3.75	4.37	4.49	4.65	4.52	4.77	5.02	5.27	4.95	4.94	4.99	5.07	5.05	5.01
7000	2.14	3.19	3.81	4.41	4.63	4.91	4.60	4.64	4.92	5.19	5.13	5.13	5.17	5.25	5.27	5.24
8000	2.14	3.21	3.90	4.42	4.72	5.00	4.68	4.76	5.02	5.28	5.21	5.21	5.23	5.31	5.33	5.36
9000	2.13	3.21	3.87	4.43	4.75	5.02	4.74	4.85	5.10	5.40	5.32	5.29	5.33	5.28	5.37	5.42
10000	2.11	3.13	3.87	4.46	4.80	5.11	4.93	5.06	5.31	5.61	5.54	5.54	5.52	5.56	5.60	5.63
Algorithm: Doolittle																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.84	2.33	2.21	1.50	1.49	1.55	1.61	1.70	1.76	1.84	1.84	1.95	1.97	1.97	2.07	2.10
4000	1.94	2.58	2.58	1.76	1.69	1.77	1.85	1.99	2.11	2.17	2.23	2.33	2.40	2.48	2.54	2.56
5000	1.98	2.78	2.93	2.23	2.10	2.14	2.21	2.40	2.54	2.65	2.71	2.83	2.94	3.04	3.12	3.16
6000	1.97	2.76	2.88	2.20	2.07	2.05	2.12	2.27	2.41	2.51	2.49	2.62	2.75	2.90	3.03	3.10
7000	1.99	2.81	3.01	2.45	2.28	2.21	2.28	2.46	2.58	2.66	2.61	2.74	2.89	3.09	3.22	3.28
8000	1.99	2.84	3.14	2.53	2.31	2.28	2.34	2.50	2.64	2.75	2.66	2.71	2.88	3.07	3.21	3.27
9000	1.98	2.82	3.12	2.72	2.48	2.46	2.54	2.64	2.81	2.89	2.82	2.83	2.95	3.12	3.24	3.29
10000	1.98	2.84	3.18	2.78	2.69	2.63	2.67	2.82	2.96	3.05	3.02	3.04	3.06	3.17	3.29	3.38

Table A2. Cont.

Intel i9-12900HK																
Algorithm: Crout																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Windows	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.04	2.91	3.54	3.70	3.89	3.98	4.02	3.95	3.92	3.95	3.92	4.09	4.23	4.31	4.31	4.39
4000	2.12	3.11	3.69	4.11	4.31	4.48	4.59	4.14	4.35	4.44	4.50	4.66	4.73	4.83	4.73	4.79
5000	2.12	3.16	3.82	4.48	4.85	5.19	4.79	4.90	5.27	5.49	5.36	5.39	5.54	5.66	5.58	5.49
6000	1.62	2.71	3.61	4.27	4.57	4.94	4.96	4.44	4.74	4.89	4.94	5.05	5.21	5.27	5.20	5.23
7000	1.60	2.71	3.68	4.31	4.72	4.95	4.96	4.49	4.83	5.03	5.02	5.15	5.11	5.18	5.18	5.16
8000	1.61	2.75	3.62	4.13	4.42	4.75	4.36	4.56	4.94	5.27	4.98	5.00	4.98	4.88	5.05	5.17
9000	1.45	2.49	3.47	4.06	4.39	4.73	4.52	4.51	4.71	4.92	4.92	4.93	4.93	4.98	5.00	5.04
10000	2.00	3.03	3.69	4.30	4.63	4.95	4.86	4.86	5.07	5.28	5.24	5.24	5.25	5.30	5.33	5.34

Table A3. Speedup results on Ryzen 7 7800X3D. Values grouped by algorithm and operating system. Bold values indicate the maximum speedup achieved for each matrix size, identifying the most performant thread configuration.

Ryzen 7 7800X3D																
Algorithm: Crout																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.83	4.71	4.71	5.65	4.71	4.71	6.33	7.07	7.07	7.07	7.07	7.07	7.07	7.07	7.07	7.07
4000	3.98	5.75	7.39	8.21	7.39	7.39	8.57	8.62	8.62	8.62	8.62	8.62	8.62	8.62	8.67	8.82
5000	4.27	6.42	7.77	8.85	8.11	8.61	9.35	10.05	9.58	9.66	9.48	9.53	9.64	9.66	9.80	10.05
6000	4.87	7.38	9.52	10.79	10.27	10.88	11.82	12.27	11.65	11.85	11.61	11.72	11.92	12.03	12.23	12.31
7000	3.88	6.71	8.70	10.17	9.96	10.41	11.25	11.56	10.61	11.00	10.66	10.62	10.85	11.03	11.11	10.92
8000	1.85	3.35	4.65	5.78	4.47	5.13	5.85	6.20	5.35	5.39	5.49	5.47	5.57	5.79	6.05	6.14
9000	1.90	3.54	4.92	6.16	4.84	5.47	6.24	6.68	5.78	5.74	5.85	5.86	5.93	6.14	6.39	6.49
10000	1.90	3.51	5.07	6.23	5.04	5.67	6.44	6.89	6.04	5.96	6.03	6.06	6.15	6.35	6.56	6.65
Windows	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.64	3.36	4.62	4.62	3.08	3.70	3.70	4.62	3.70	3.78	4.62	4.62	3.70	3.70	4.62	4.40
4000	0.99	2.08	3.23	4.10	3.55	4.07	4.56	5.07	3.99	4.10	4.26	4.47	4.32	4.60	4.92	4.95
5000	0.93	1.94	2.88	3.67	3.48	3.78	4.36	4.75	3.55	3.47	3.66	3.86	4.04	4.34	4.62	4.81
6000	0.95	1.91	2.85	3.74	4.24	4.97	5.72	6.26	4.79	4.61	4.75	5.01	5.27	5.65	6.05	6.24
7000	1.04	2.04	3.04	3.98	4.69	5.53	6.34	6.98	5.38	5.18	5.29	5.55	5.53	6.04	6.36	6.52
8000	1.00	1.86	2.70	3.52	4.23	4.96	5.68	6.19	4.94	4.78	4.79	4.93	5.16	5.47	5.73	5.87
9000	1.24	3.42	4.70	4.30	4.75	5.41	6.08	5.86	5.44	5.44	5.47	5.51	5.73	6.03	6.32	6.38
10000	1.87	3.52	4.90	5.76	4.78	5.46	6.12	6.47	5.51	5.48	5.52	5.58	5.78	6.10	6.34	6.33
Algorithm: Doolittle																
Linux	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.76	2.76	3.53	4.57	2.69	2.82	3.44	3.47	3.44	3.50	3.50	3.47	3.47	3.44	3.47	3.50
4000	1.91	3.14	4.27	4.91	3.41	3.91	4.34	4.72	4.30	4.37	4.42	4.39	4.33	4.50	4.73	4.88
5000	1.97	3.23	4.37	5.18	3.53	3.94	4.43	4.73	4.15	4.28	4.36	4.31	4.30	4.51	4.71	4.97
6000	1.95	3.40	4.69	5.77	4.21	4.84	5.51	5.92	5.03	5.21	5.30	5.22	5.27	5.55	5.87	6.09
7000	1.91	3.46	4.74	5.84	4.42	5.06	5.78	6.15	5.25	5.42	5.47	5.42	5.48	5.69	6.01	6.18
8000	1.87	3.36	4.68	5.78	4.47	5.12	5.83	6.23	5.38	5.37	5.49	5.46	5.55	5.79	6.05	6.20
9000	1.89	3.49	4.97	6.09	4.81	5.43	6.20	6.67	5.76	5.73	5.81	5.84	5.93	6.11	6.37	6.44
10000	1.89	3.61	5.03	6.30	5.05	5.64	6.44	6.92	6.00	5.96	6.07	6.06	6.15	6.34	6.60	6.63
Windows	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.89	2.90	4.25	4.25	2.83	3.40	3.40	3.83	3.40	3.40	3.45	4.15	3.40	3.40	3.59	4.21
4000	1.94	3.29	3.07	3.51	3.23	3.71	4.19	4.71	3.62	3.60	3.71	3.92	3.87	4.16	4.34	4.68
5000	0.87	1.78	2.62	3.33	3.26	3.60	4.05	4.45	3.44	3.30	3.42	3.63	3.82	4.02	4.32	4.50
6000	1.06	3.56	4.82	5.65	4.35	5.00	5.60	6.09	5.19	5.24	5.16	5.25	5.36	5.70	6.03	6.22
7000	1.97	2.28	2.91	3.80	4.49	5.26	6.02	6.66	5.21	5.05	5.11	5.33	5.59	5.94	6.27	6.46
8000	1.08	2.06	3.03	3.95	4.76	5.61	6.42	6.97	5.59	5.39	5.43	5.62	5.88	6.22	6.55	6.71
9000	1.70	3.61	4.98	5.92	5.07	5.77	6.46	6.95	5.79	5.77	5.79	5.86	6.05	6.40	6.71	6.83
10000	1.90	3.55	4.85	5.55	5.02	5.70	6.41	6.69	5.66	5.66	5.67	5.73	5.91	6.19	6.45	6.44

Table A4. Speedup results on Intel Xeon Phi 7250. Values grouped by algorithm and operating system. Bold values indicate the maximum speedup achieved for each matrix size, identifying the most performant thread configuration.

Intel Xeon Phi 7250																	
Algorithm: Crout																	
Linux	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272
3000	9.70	12.94	13.44	13.43	13.98	14.17	14.03	14.11	14.17	14.15	14.06	13.68	13.47	13.60	13.46	13.35	12.93
4000	11.03	16.59	16.11	17.02	17.68	18.12	18.52	18.65	18.76	18.62	18.59	18.61	18.62	18.61	18.41	17.98	17.54
5000	10.72	14.42	14.49	15.06	16.01	16.46	16.23	16.20	16.62	16.66	16.58	16.25	15.87	15.40	15.08	14.58	14.44
6000	12.45	20.13	21.82	23.22	24.58	25.25	24.75	24.66	24.47	24.59	24.85	25.46	25.74	25.47	25.59	24.97	24.64
7000	13.70	25.53	29.16	32.14	34.28	35.92	35.66	35.77	35.47	35.32	34.99	35.49	35.61	35.83	36.39	36.87	37.18
8000	15.64	30.41	36.60	41.25	43.65	47.58	48.29	48.58	48.95	48.99	48.13	47.24	47.10	47.36	47.43	48.33	48.65
9000	15.14	27.98	27.46	30.79	33.09	36.52	38.81	39.53	40.13	40.88	40.39	40.09	39.88	39.86	39.42	38.85	38.69
10000	15.19	29.64	39.87	45.62	50.07	56.18	62.00	64.53	65.88	66.71	65.89	66.13	67.22	67.71	67.47	67.18	65.83
Algorithm: Doolittle																	
Linux	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272
3000	9.76	13.02	13.62	13.62	13.95	13.78	13.78	13.62	13.16	12.60	12.08	11.83	11.60	11.26	12.46	11.49	10.65
4000	10.92	16.74	16.08	17.27	17.27	17.64	18.03	18.44	18.13	17.00	16.83	16.74	16.32	15.77	16.16	15.77	15.18
5000	10.93	14.80	15.17	15.52	16.40	16.86	16.45	16.18	16.72	16.81	16.81	16.23	15.68	16.49	15.13	15.52	14.73
6000	12.07	19.16	21.58	24.01	25.34	25.17	23.54	24.46	23.29	21.94	22.78	24.81	24.16	24.19	25.01	23.97	24.19
7000	13.41	25.14	28.07	31.46	32.96	35.05	34.59	34.59	34.36	33.94	33.72	34.36	34.39	35.05	35.29	35.56	36.41
8000	15.94	30.50	36.73	40.46	43.54	46.86	47.67	47.89	48.20	48.36	47.95	47.28	46.65	46.88	47.49	47.64	48.42
9000	14.59	27.29	26.52	29.44	32.14	35.50	37.31	38.12	38.58	39.08	38.80	38.54	38.41	38.27	37.97	37.42	37.34
10000	15.52	30.59	40.94	46.90	50.68	56.45	62.20	63.56	64.34	65.79	65.53	65.30	66.21	66.00	65.75	64.82	63.31

Appendix B. Detailed Energy Consumption Results

This appendix details the whole-system energy consumption, measured in kilojoules (kJ), for all experimental configurations. The data cover sequential and parallel executions for different matrix sizes and thread counts on the four studied architectures. In these tables, the minimum energy consumption value for each row is marked in bold, representing the most energy-efficient configuration for that specific scenario.

Table A5. Energy consumption (kJ) on Mac Mini M4. Values grouped by algorithm and operating system. Bold values represent the minimum energy consumption for each matrix size.

Mac Mini M4																	
Algorithm: Crout																	
macOS	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	0.15	0.15	0.12	0.09	0.09	0.09	0.08	0.11	0.11	0.10	0.12	0.09	0.12	0.09	0.10	0.12	0.10
4000	0.59	0.61	0.42	0.41	0.36	0.38	0.37	0.34	0.36	0.35	0.37	0.39	0.39	0.39	0.38	0.38	0.35
5000	1.36	1.19	1.01	0.94	0.85	0.82	0.83	0.83	0.88	0.86	0.88	0.93	0.89	0.89	0.86	0.90	0.88
6000	3.04	2.13	1.89	1.74	1.72	1.61	1.61	1.54	1.64	1.63	1.72	1.73	1.69	1.72	1.67	1.69	1.66
7000	4.69	3.52	3.19	2.86	2.75	2.64	2.67	2.55	2.72	2.69	2.81	2.81	2.89	2.83	2.81	2.80	2.76
8000	7.30	5.22	4.84	4.44	4.21	4.02	4.12	4.02	4.28	4.15	4.36	4.40	4.44	4.34	4.30	4.39	4.29
9000	9.74	7.58	7.09	6.42	6.06	5.87	6.04	5.84	6.29	6.17	6.50	6.53	6.52	6.48	6.37	6.40	6.36
10000	13.64	10.63	9.85	9.07	8.56	8.28	8.57	8.35	8.83	8.96	9.13	9.22	9.27	9.19	9.08	9.09	9.05
Algorithm: Doolittle																	
macOS	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	0.21	0.16	0.15	0.15	0.10	0.10	0.14	0.15	0.15	0.15	0.13	0.09	0.15	0.16	0.15	0.11	0.16
4000	0.61	0.55	0.45	0.44	0.45	0.45	0.44	0.43	0.40	0.45	0.43	0.46	0.42	0.40	0.46	0.45	0.40
5000	1.38	1.21	1.03	0.98	0.95	0.89	0.88	0.87	0.91	0.90	0.94	0.96	0.96	0.94	0.94	0.93	0.93
6000	2.59	2.18	1.96	1.79	1.73	1.61	1.57	1.58	1.72	1.70	1.71	1.77	1.76	1.77	1.72	1.76	1.71
7000	4.38	3.52	3.19	2.98	2.86	2.69	2.68	2.62	2.75	2.73	2.83	2.91	2.87	2.82	2.81	2.82	2.79
8000	6.74	5.28	4.88	4.48	4.27	4.07	4.13	4.06	4.22	4.19	4.36	4.43	4.46	4.39	4.34	4.43	4.37
9000	9.79	7.62	7.07	6.46	6.15	5.95	6.01	5.96	6.27	6.26	6.46	6.51	6.56	6.44	6.44	6.45	6.37
10000	13.63	10.66	9.77	9.01	8.54	8.30	8.54	8.35	8.87	8.93	9.14	9.22	9.23	9.17	9.05	9.14	9.05

Table A6. Energy consumption (kJ) on Intel Core i9-12900HK. Values grouped by algorithm and operating system. Bold values represent the minimum energy consumption for each matrix size.

Intel i9-12900HK																	
Algorithm: Crout																	
Linux	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.41	1.00	0.77	0.90	1.16	1.30	1.20	1.12	1.10	1.19	1.11	1.11	1.07	0.93	1.04	0.96	1.05
4000	4.09	2.72	2.26	2.07	3.16	3.13	3.28	3.08	2.89	2.96	2.70	2.74	2.45	2.47	2.48	2.43	2.28
5000	9.44	6.14	4.83	4.55	6.32	6.82	6.54	6.30	5.89	5.60	5.20	5.13	5.01	4.80	4.67	4.51	4.49
6000	16.49	10.79	8.20	7.75	10.45	11.28	11.33	11.13	10.19	9.69	9.68	9.68	9.21	8.54	8.13	7.78	7.61
7000	28.57	18.36	13.37	13.07	15.90	17.82	18.11	17.73	16.53	16.30	15.33	15.47	15.62	14.28	13.14	12.54	12.81
8000	45.23	28.96	21.48	20.02	24.47	27.54	28.52	27.57	24.75	24.74	23.51	24.76	23.14	21.78	20.70	19.87	19.97
9000	70.57	45.83	32.75	30.20	33.52	39.13	40.46	37.96	37.22	35.86	34.47	35.53	35.88	34.50	31.42	29.77	29.02
10000	106.86	65.45	45.33	41.97	50.52	49.93	53.27	52.21	47.89	48.10	47.99	46.75	48.30	47.71	44.85	45.20	44.00
Windows	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.18	1.32	1.06	1.01	1.00	0.81	0.89	0.89	1.87	0.85	0.77	0.83	0.82	0.73	0.68	0.82	0.88
4000	5.76	3.96	2.69	2.31	2.23	2.14	2.26	2.16	2.12	1.80	2.12	1.97	2.08	1.90	1.85	2.06	1.93
5000	14.79	9.26	6.43	5.23	4.63	4.37	4.25	4.09	4.16	3.99	3.75	3.55	3.69	3.68	3.44	3.55	3.93
6000	23.90	15.11	10.32	8.31	7.03	7.32	7.28	7.62	8.08	6.82	6.42	7.02	7.20	6.38	6.47	6.20	6.87
7000	44.15	25.66	17.83	15.22	13.69	12.06	12.52	12.84	12.67	11.18	11.21	10.72	11.30	11.13	10.74	10.70	11.11
8000	69.99	40.56	27.08	22.74	20.83	17.91	17.74	19.78	19.24	16.90	16.43	16.88	16.51	17.66	16.46	16.61	16.38
9000	102.28	60.44	43.48	32.73	30.61	27.73	26.75	28.81	28.55	25.77	25.64	24.78	24.99	25.07	25.53	24.58	24.36
10000	152.15	92.40	61.25	49.89	44.73	41.49	39.33	39.96	39.10	37.25	35.44	35.82	35.67	36.34	35.71	34.28	37.62
Algorithm: Doolittle																	
Linux	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.37	0.86	0.79	0.88	1.05	1.31	1.10	1.16	1.10	1.18	1.07	0.94	0.98	0.88	1.07	1.00	0.96
4000	4.19	2.73	2.09	2.11	3.06	3.06	3.01	2.90	2.97	2.73	2.64	2.83	2.47	2.61	2.29	2.30	2.11
5000	9.90	6.08	4.73	4.33	6.43	6.72	6.50	5.76	5.62	5.61	5.21	5.05	4.94	4.62	4.68	4.56	4.53
6000	16.70	10.65	8.32	7.66	11.03	11.71	11.03	10.33	10.79	9.93	9.24	9.65	8.86	8.21	8.16	7.71	7.48
7000	28.27	18.39	13.64	13.01	16.21	17.69	18.23	17.71	16.58	15.77	15.28	16.17	15.31	14.06	12.70	12.21	11.02
8000	45.87	27.93	20.13	19.59	23.88	27.22	28.25	26.73	24.89	24.17	22.75	24.12	23.68	21.74	19.55	19.23	17.70
9000	70.72	43.20	32.69	30.64	34.92	38.09	39.62	38.34	37.45	35.24	34.65	34.73	36.03	31.83	31.43	29.16	29.22
10000	108.02	65.97	47.72	42.86	49.19	52.67	54.13	53.34	52.00	50.13	46.34	47.05	48.05	46.78	45.81	42.70	41.38
Windows	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.88	1.39	1.04	0.92	0.97	0.85	0.85	0.81	0.78	0.90	0.83	0.79	0.76	0.91	0.85	0.82	0.81
4000	6.08	3.82	2.63	2.38	2.29	2.40	2.10	2.11	2.21	2.05	1.86	1.85	1.98	1.99	1.94	1.92	2.07
5000	15.74	9.39	6.23	5.07	4.31	4.20	4.23	4.42	4.45	3.84	3.73	3.49	3.99	3.92	3.85	3.79	4.02
6000	25.40	15.36	10.43	8.94	8.00	7.70	6.92	7.96	7.92	7.57	7.31	7.10	7.09	6.96	6.73	6.50	7.17
7000	44.69	26.55	17.95	14.31	13.79	12.53	12.01	12.49	13.63	11.96	10.92	11.15	11.64	10.86	11.53	10.61	11.26
8000	69.30	40.33	28.25	23.61	20.31	19.18	18.01	19.95	19.64	16.94	16.64	17.37	16.96	17.14	16.64	17.11	16.65
9000	105.29	62.52	41.23	36.58	30.73	28.74	27.45	28.04	28.62	26.84	25.37	26.89	25.50	25.69	25.23	25.73	25.64
10000	157.91	91.11	60.88	49.65	45.56	42.87	40.61	42.74	39.89	38.16	36.05	36.46	37.45	35.55	37.77	35.38	35.08

Table A7. Energy consumption (kJ) on Ryzen 7 7800X3D. Values grouped by algorithm and operating system. Bold values represent the minimum energy consumption for each matrix size.

Ryzen 7 7800X3D																	
Algorithm: Crout																	
Linux	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.01	1.58	1.04	1.14	1.28	1.28	1.07	1.33	1.61	1.40	1.25	1.36	0.97	1.58	1.57	1.09	1.43
4000	6.40	3.89	2.66	2.21	2.07	2.52	2.60	2.28	2.31	2.37	2.40	2.21	2.18	2.27	2.12	1.93	2.19
5000	14.77	8.30	4.71	3.82	3.51	4.48	3.41	3.80	3.63	4.23	4.04	4.46	4.05	4.74	4.29	3.57	3.70
6000	33.25	18.33	10.53	8.34	6.84	8.96	7.78	7.30	7.11	8.29	8.20	8.04	8.03	7.41	7.38	7.50	6.70
7000	56.92	31.86	18.33	14.37	12.12	15.66	13.52	12.80	12.01	13.84	12.96	12.65	12.76	12.93	13.03	12.46	12.52
8000	93.64	52.21	31.75	23.56	20.55	24.46	21.81	19.99	18.79	20.84	21.59	19.93	20.61	21.25	20.32	18.84	19.40
9000	141.87	81.44	45.83	34.96	28.86	35.95	32.85	29.56	28.36	31.48	30.00	30.32	31.02	29.65	28.93	28.14	28.52
10000	212.93	118.40	66.08	51.25	41.31	50.68	43.65	40.63	38.80	42.44	43.89	44.13	45.63	43.69	41.88	40.12	41.99

Table A7. Cont.

Ryzen 7 7800X3D																	
Algorithm: Crout																	
Linux	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Windows	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	1.41	1.65	1.31	0.90	1.16	1.14	1.05	0.92	1.01	0.90	0.94	1.24	0.79	0.92	1.14	1.24	1.14
4000	5.79	7.83	4.80	2.90	2.57	1.89	1.86	2.78	3.53	3.81	4.57	3.93	3.62	3.62	3.92	3.64	3.90
5000	12.88	14.74	8.43	5.46	4.34	4.65	3.97	3.84	3.92	4.04	3.90	3.93	4.33	4.14	3.60	3.39	3.55
6000	29.30	32.33	18.14	12.37	10.95	8.93	7.87	7.31	6.22	7.68	7.58	7.53	7.37	7.50	7.16	7.22	6.34
7000	54.39	53.97	30.71	21.98	17.16	14.76	11.94	11.46	11.17	12.44	13.82	13.43	12.37	12.44	11.56	10.74	11.23
8000	91.38	84.09	47.09	34.40	27.77	22.48	19.27	18.15	18.51	19.55	20.97	21.87	20.78	19.75	18.52	18.27	17.91
9000	133.89	122.79	68.56	50.21	40.74	32.30	28.66	26.98	25.52	29.12	30.50	31.03	30.78	28.96	27.11	25.75	26.52
10000	192.00	169.30	98.76	71.09	55.79	48.24	42.70	37.44	36.76	42.67	42.59	41.17	43.07	43.06	39.40	38.44	40.46
Algorithm: Doolittle																	
Linux	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.06	1.58	1.28	0.98	1.08	1.09	1.05	0.91	1.23	0.93	0.93	1.31	0.90	0.95	0.85	0.88	1.21
4000	6.68	3.78	2.49	2.25	2.04	2.52	2.23	1.96	2.22	1.78	2.14	1.90	1.94	2.29	1.99	1.76	2.00
5000	14.59	7.53	5.27	3.92	3.54	4.43	4.40	4.01	3.96	4.11	4.05	3.83	4.36	4.16	4.12	3.66	4.06
6000	33.64	18.31	11.17	8.13	7.41	9.49	8.87	7.60	7.08	8.58	7.80	8.27	8.03	7.47	7.58	7.29	7.24
7000	57.55	31.68	19.02	14.78	12.50	15.05	14.18	12.43	11.99	12.98	13.12	12.29	13.53	13.62	12.31	12.65	12.35
8000	91.47	50.69	30.41	23.89	18.77	24.52	22.73	19.99	19.15	20.35	20.52	21.28	20.84	20.04	20.27	19.12	19.99
9000	145.73	78.94	47.19	34.94	28.53	35.47	32.47	28.69	27.95	31.20	30.96	31.18	29.76	31.43	30.08	29.06	29.36
10000	216.38	118.64	70.71	52.66	42.94	50.65	45.98	42.58	41.13	43.91	43.65	44.90	45.10	42.13	41.46	41.96	40.50
Windows	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
3000	2.42	0.88	1.65	1.34	1.00	0.82	0.66	0.91	0.87	0.78	0.84	0.96	0.83	0.76	0.92	0.85	0.94
4000	6.86	7.81	4.67	3.01	2.55	2.35	1.94	1.68	1.72	2.06	1.96	2.20	2.33	2.06	2.22	1.99	1.89
5000	13.63	14.94	8.58	5.95	4.85	4.75	4.43	3.88	3.50	4.47	4.69	4.64	4.11	3.72	3.47	3.67	3.69
6000	32.34	30.97	17.17	12.24	9.28	7.96	7.13	7.25	7.04	7.45	7.72	8.18	7.65	6.34	7.27	6.52	6.64
7000	58.74	51.93	28.42	20.46	16.57	14.38	13.30	11.32	10.24	12.98	13.21	12.28	11.97	11.46	11.32	11.04	11.72
8000	94.36	83.54	47.40	34.01	27.02	22.90	20.48	18.29	18.25	20.42	20.69	20.00	20.94	20.41	18.59	17.98	18.13
9000	142.73	125.12	69.50	51.43	40.60	33.75	29.90	28.22	26.13	30.27	29.89	30.64	30.56	29.48	28.70	28.65	28.04
10000	200.37	179.70	102.52	73.70	58.62	49.68	43.20	42.22	39.03	42.92	45.44	42.70	45.06	43.82	42.46	41.23	40.24

Table A8. Energy consumption (kJ) on Intel Xeon Phi 7250. Values grouped by algorithm and operating system. Bold values represent the minimum energy consumption for each matrix size.

Intel Xeon Phi 7250																		
Algorithm: Crout																		
Linux	1	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272
3000	63.36	7.47	5.93	5.88	5.71	5.95	5.41	5.46	5.54	5.48	5.56	5.84	5.79	6.08	5.84	6.06	6.45	6.10
4000	167.93	18.81	12.95	13.13	12.87	12.46	11.94	11.84	11.75	11.92	11.42	12.58	12.98	12.21	12.97	12.61	12.31	13.00
5000	330.86	32.83	27.21	26.06	25.24	24.45	23.75	24.14	23.42	23.88	24.11	23.68	24.25	25.16	25.56	24.32	24.72	24.59
6000	795.61	70.36	54.47	49.05	42.68	40.70	43.71	45.11	45.73	42.00	44.20	39.33	43.20	42.05	42.45	40.06	40.00	42.20
7000	2027.27	169.05	97.68	85.40	77.52	71.80	69.52	69.25	68.90	69.60	69.29	70.57	69.31	71.39	70.60	67.77	67.34	67.64
8000	4477.49	296.23	160.44	137.08	118.09	113.65	102.75	99.05	100.37	99.05	97.33	100.68	100.95	104.64	102.80	102.59	98.44	102.69
9000	5058.30	353.20	205.82	216.31	191.13	179.36	163.73	152.46	148.45	144.30	145.98	145.31	145.74	147.62	148.85	151.79	152.76	155.03
10000	12440.61	802.69	439.61	336.24	294.29	257.74	237.12	221.64	212.66	202.40	204.59	205.39	206.98	205.22	204.95	206.70	209.37	215.28
Algorithm: Doolittle																		
Linux	1	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272
3000	63.42	7.46	5.93	5.62	5.74	5.61	5.71	5.60	5.55	5.57	5.70	5.99	5.80	6.23	6.20	6.13	6.11	6.23
4000	168.29	18.29	12.91	13.17	12.56	12.36	12.32	12.08	11.70	12.31	12.48	13.02	12.17	12.99	12.74	12.34	12.90	13.08
5000	324.86	33.53	26.45	26.05	24.93	25.20	23.95	24.68	25.29	24.31	23.44	24.10	23.05	23.27	24.70	25.31	25.50	23.91
6000	899.90	78.86	51.40	47.21	43.27	44.08	40.03	40.62	40.73	41.78	43.70	42.09	39.35	40.38	39.21	38.83	41.63	42.33
7000	1964.62	163.89	97.90	85.54	78.44	72.62	68.27	68.96	70.66	67.28	69.70	70.00	70.29	70.04	67.38	68.91	68.01	67.42
8000	4073.93	280.58	157.56	131.68	118.19	114.98	102.59	100.63	100.33	98.35	97.53	100.94	101.12	102.02	100.93	102.68	103.97	102.03
9000	4904.13	351.85	212.26	218.75	193.73	177.00	160.46	154.56	149.10	148.63	145.00	144.46	147.99	147.88	147.52	149.54	155.18	155.35
10000	11904.53	803.16	431.77	337.74	292.34	261.89	237.13	220.12	210.94	203.82	206.43	204.39	207.69	202.83	204.25	204.71	212.19	216.07

Appendix C. Performance–Energy Trade-Offs Using Spearman Rank Correlation

This appendix presents the results of the Spearman rank correlation analysis between execution time and energy consumption for each architecture–algorithm–operating system combination. These coefficients quantify how strongly performance gains are associated

with energy savings. The values are calculated at a 95% confidence level, and significant findings that characterize the performance–energy trade-offs are highlighted in bold to facilitate the interpretation of the results.

Table A9. Spearman correlation coefficients and corresponding p-values between runtime and energy consumption for each configuration and matrix size. Bold values indicate significant correlations ($p < 0.05$) between execution time and energy consumption.

			3000	4000	5000	6000	7000	8000	9000	10,000
Mac Mini M4	macOS	Crout	0.4602 0.063	0.7639 3.57×10^{-04}	0.8640 7.78×10^{-06}	0.9381 2.66×10^{-08}	0.8866 2.13×10^{-06}	0.9240 1.18×10^{-07}	0.9559 2.20×10^{-09}	0.9167 2.31×10^{-07}
		Doolittle	0.5842 0.014	0.1934 0.457	0.5387 0.026	0.6413 0.006	0.5469 0.023	0.6422 0.005	0.6683 0.003	0.7399 6.85×10^{-04}
Intel i9-12900HK	Linux	Crout	0.9037 6.57×10^{-07}	0.9314 5.64×10^{-08}	0.9804 5.40×10^{-12}	0.9681 1.99×10^{-10}	0.9387 2.46×10^{-08}	0.9265 9.32×10^{-08}	0.9020 7.50×10^{-07}	0.9118 3.50×10^{-07}
		Doolittle	0.7786 2.31×10^{-04}	0.9589 1.30×10^{-09}	0.9510 4.79×10^{-09}	0.9479 7.51×10^{-09}	0.9877 1.62×10^{-13}	0.9853 6.34×10^{-13}	0.9608 9.24×10^{-10}	0.9755 2.84×10^{-11}
	Windows	Crout	0.7162 0.001	0.7652 3.44×10^{-04}	0.9412 1.82×10^{-08}	0.9118 3.50×10^{-07}	0.9387 2.46×10^{-08}	0.9632 5.74×10^{-10}	0.9534 3.28×10^{-09}	0.9191 1.86×10^{-07}
		Doolittle	0.6888 0.002	0.7860 1.83×10^{-04}	0.8584 1.03×10^{-05}	0.8750 4.28×10^{-06}	0.9093 4.27×10^{-07}	0.9104 3.90×10^{-07}	0.9216 1.49×10^{-07}	0.9700 1.28×10^{-10}
	Linux	Crout	0.0374 0.887	0.6901 0.002	0.4589 0.064	0.7868 1.79×10^{-04}	0.7475 5.61×10^{-04}	0.9559 2.20×10^{-09}	0.9681 1.99×10^{-10}	0.9191 1.86×10^{-07}
		Doolittle	0.2516 0.330	0.6985 0.002	0.8309 3.61×10^{-05}	0.9461 9.65×10^{-09}	0.8768 3.86×10^{-06}	0.9240 1.18×10^{-07}	0.9265 9.32×10^{-08}	0.9583 1.44×10^{-09}
Ryzen 7 7800X3D	Windows	Crout	0.3720 0.141	0.2601 0.313	0.8235 4.85×10^{-05}	0.9681 1.99×10^{-10}	0.9847 8.63×10^{-13}	0.9706 1.10×10^{-10}	0.9142 2.85×10^{-07}	0.8603 9.42×10^{-06}
		Doolittle	0.1028 0.695	0.8179 6.04×10^{-05}	0.9559 2.20×10^{-09}	0.7745 2.62×10^{-04}	0.9559 2.20×10^{-09}	0.9461 9.65×10^{-09}	0.9050 5.99×10^{-07}	0.9412 1.82×10^{-08}
Intel Xeon Phi 7250	Linux	Crout	0.8353 1.60×10^{-05}	0.7324 5.47×10^{-04}	0.8679 3.05×10^{-06}	0.5996 0.009	0.8844 1.10×10^{-06}	0.9236 4.56×10^{-08}	0.9814 6.72×10^{-13}	0.8947 5.42×10^{-07}
		Doolittle	0.8702 2.68×10^{-06}	0.8052 5.55×10^{-05}	0.6343 0.005	0.7021 0.001	0.8444 1.04×10^{-05}	0.8369 1.48×10^{-05}	0.9711 2.23×10^{-11}	0.9443 3.92×10^{-09}

References

1. Golub, G.H.; Van Loan, C.F. *Matrix Computations*, 4 ed.; Johns Hopkins University Press: Baltimore, MD, USA, 2013.
2. Trefethen, L.N.; Bau, D. *Numerical Linear Algebra*; SIAM: Philadelphia, PA, USA, 2022.
3. Dongarra, J.J.; Duff, I.S.; Sorensen, D.C.; Van der Vorst, H.A. *Numerical Linear Algebra for High-Performance Computers*; SIAM: Philadelphia, PA, USA, 1998.
4. Fang, J.; Huang, C.; Tang, T.; Wang, Z. Parallel programming models for heterogeneous many-cores: A comprehensive survey. *CCF Trans. High Perform. Comput.* **2020**, *2*, 382–400.
5. Aragón-Jurado, J.M.; Bangash, A.A.; Dorrnsoro, B.; Ali, K.; Hindle, A.; Ruiz, P. Does faster mean greener? Runtime and energy trade-offs in iOS applications with compiler optimizations. *Sustain. Comput. Inform. Syst.* **2025**, *47*, 101166.
6. Peng, J.; Li, K.; Chen, J.; Li, K. Reliability/performance-aware scheduling for parallel applications with energy constraints on heterogeneous computing systems. *IEEE Trans. Sustain. Comput.* **2022**, *7*, 681–695.
7. Crout, P.D. A Short Method for Evaluating Determinants and Solving Systems of Linear Equations with Real or Complex Coefficients. *Electr. Eng.* **1941**, *60*, 1235–1240.
8. Doolittle, M.H. Method employed in the solution of normal equations and the adjustment of a triangulation. *US Coast Geod. Surv. Rep.* **1878**, *1878*, 115–120.
9. de Falguerolles, A. Cholesky and the Cholesky decomposition: A commemoration by an applied statistician. *J. Soc. Stat.* **2019**, *160*, 83–96.
10. Higham, N.J.; Higham, D.J. Large Growth Factors in Gaussian Elimination with Pivoting. *SIAM J. Matrix Anal. Appl.* **1989**, *10*, 155–164.
11. Higham, N.J. *Accuracy and Stability of Numerical Algorithms*, 2nd ed.; SIAM: Philadelphia, PA, USA, 2002.
12. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; et al. *LAPACK Users' Guide*, 3rd ed.; SIAM: Philadelphia, PA, USA, 1999.
13. Blackford, S.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; Petitet, A.; Stanley, K.; Walker, D.; White, R. *ScaLAPACK Users' Guide*; Technical report; Lawrence Berkeley National Laboratory/University of Tennessee: Knoxville, TN, USA, 1997.

14. Lindquist, N.; Luszczek, P.; Dongarra, J. Using Additive Modifications in LU Factorization Instead of Pivoting. In Proceedings of the 37th ACM International Conference on Supercomputing (ICS), Orlando, FL, USA, 21–23 June 2023; ACM: New York, NY, USA, 2023. <https://doi.org/10.1145/3577193.3593731>.
15. Kaya, D.; Wright, K. Parallel Algorithms for LU Decomposition on a Shared Memory Multiprocessor. *Appl. Math. Comput.* **2005**, *163*, 179–191. <https://doi.org/10.1016/j.amc.2004.01.027>.
16. Abeles, P.; contributors. EJML—Efficient Java Matrix Library. Available online: <https://ejml.org> (accessed on 2 October 2025).
17. Wendykier, P.; Nagy, J.G. Parallel Colt: A High-Performance Java Library for Scientific Computing and Image Processing. *ACM Trans. Math. Softw.* **2010**, *37*, 31. <https://doi.org/10.1145/1824801.1824809>
18. Basso, M.; Rosà, A.; Omini, L.; Binder, W. Java Vector API: Benchmarking and Performance Analysis. In Proceedings of the Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, Montréal, QC, Canada, 25–26 February 2023; pp. 1–12. <https://doi.org/10.1145/3578360.3580265>.
19. Blumofe, R.D.; Leiserson, C.E. Scheduling multithreaded computations by work stealing. *J. ACM (JACM)* **1999**, *46*, 720–748.
20. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 5.0. 2018. Available online: <https://www.openmp.org/specifications/> (accessed on 1 January 2026).
21. Bircher, W.L.; John, L.K. Analysis of dynamic power management on multi-core processors. In Proceedings of the 22nd Annual International Conference on Supercomputing, Island of Kos, Greece, 7–12 June 2008; pp. 327–338.
22. Rountree, B.; Lownenthal, D.K.; De Supinski, B.R.; Schulz, M.; Freeh, V.W.; Bletsch, T. Adagio: Making DVS practical for complex HPC applications. In Proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, 8–12 June 2009; pp. 460–469.
23. Freeh, V.W.; Lowenthal, D.K. Using multiple energy gears in MPI programs on a power-scalable cluster. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA, 15–17 June 2005; pp. 164–173.
24. Esmaeilzadeh, H.; Blem, E.; Amant, R.S.; Sankaralingam, K.; Burger, D. Power challenges may end the multicore era. *Commun. ACM* **2013**, *56*, 93–102.
25. Kramer, T.M.; Bartoszewicz, A.; et al. Energy-Aware Scheduling for High-Performance Computing Systems: A Survey. *Energies* **2023**, *16*, 890.
26. Verdecchia, R.; Lago, P.; Ebert, C.; De Vries, C. Green IT and green software. *IEEE Softw.* **2021**, *38*, 7–15.
27. Abera, S.; Balakrishnan, M.; Kumar, A. Performance-energy trade-off in modern CMPs. *ACM Trans. Archit. Code Optim. (TACO)* **2020**, *18*, 1–26.
28. Al Kez, D.; Foley, A.M.; Lavery, D.; Del Rio, D.F.; Sovacool, B. Exploring the sustainability challenges facing digitalization and internet data centers. *J. Clean. Prod.* **2022**, *371*, 133633.
29. Mocnej, J.; Miškuf, M.; Papcun, P.; Zolotová, I. Impact of edge computing paradigm on energy consumption in IoT. *IFAC-PapersOnLine* **2018**, *51*, 162–167.
30. Żytko, D.; Badurowicz, M. Analysis of Performance and Energy Efficiency of Processors with Hybrid Architecture. *J. Comput. Sci. Inst.* **2025**, *34*, 54–59. <https://doi.org/10.35784/jcsi.6690>.
31. Schöne, R.; Velten, M.; Hackenberg, D.; Ilsche, T. Energy Efficiency Features of the Intel Alder Lake Architecture. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE), London, UK, 7–11 May 2024; ACM: New York, NY, USA, 2024. <https://doi.org/10.1145/3629526.3645040>.
32. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A. On the impact of code smells on the energy consumption of mobile applications. *Inf. Softw. Technol.* **2019**, *105*, 43–55.
33. Ristov, S.; Gusev, M. Performance vs Cost for Windows and Linux Platforms in Windows Azure Cloud. In Proceedings of the 2013 IEEE 2nd International Conference on Cloud Networking (CloudNet), San Francisco, CA, USA, 11–13 November 2013; pp. 194–200. <https://doi.org/10.1109/CloudNet.2013.6710581>.
34. Baggs, R.; Delgado, H.; Bakhtiani, P. Performance Analysis of I/O-Intensive & CPU-Intensive Benchmarks on Windows 7, 8.1 & Ubuntu 14.04 LTS. In Proceedings of the IEEE International Conference on Electro/Information Technology (EIT), Dekalb, IL, USA, 21–23 May 2015; pp. 415–420. <https://doi.org/10.1109/EIT.2015.7293429>.
35. Saez, J.C.; Prieto-Matias, M. Evaluation of the Intel thread director technology on an Alder Lake processor. In Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '22, New York, NY, USA, 23–24 August 2022; pp. 61–67. <https://doi.org/10.1145/3546591.3547532>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.