energies

MDPI

Article

Investigation of Energy and Power Characteristics of Various Matrix Multiplication Algorithms

Salem Alsari * and Muhammad Al-Hashimi

Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 25732, Saudi Arabia; mhashimi@kau.edu.sa
* Correspondence: ssalemalsari@stu.kau.edu.sa

Abstract: This work studied the energy behavior of six matrix multiplication algorithms with various physical asset usage patterns. Two were variants of the straight inner product of rows and columns. The rest were variants of Strassen's divide-and-conquer. Cases varied in ways that were expected to affect energy behavior. The study collected data for square matrix dimensions up to 4000. The research used reliable on-chip integrated voltage regulators embedded in a recent HPC-class AMD CPU for power measurements. Inner product methods used much less energy than the others for small to moderately large matrices. The advantage diminished for sufficiently large dimensions. The power draw of the inner product methods was less for small dimensions. After a point, the power advantage shifted significantly in favor of the divide-and-conquer group (average of 24% better), with the more block-optimized versions showing increased power efficiency (at least 8.3% better than the base method). The study explored the interplay between algorithm design, power efficiency, and computational resources. It aims to help advance the cause of power efficiency in HPC and other scenarios that rely on this vital computation.

Keywords: matrix multiplication; power efficiency; AMD EPYC; memory-optimized Strassen multiplication; power-aware algorithm; green HPC

Citation: Alsari, S.; Al-Hashimi, M. Investigation of Energy and Power Characteristics of Various Matrix Multiplication Algorithms. *Energies* **2024**, *17*, 2225. https://doi.org/10.3390/en17092225

Academic Editors: Shuaibing Li, Guoqiang Gao, Guochang Li, Yi Cui, Jiefeng Liu, Guangya Zhu and Jin Li

Received: 15 April 2024
Revised: 28 April 2024
Accepted: 28 April 2024
Published: 5 May 2024

1. Introduction

Matrix multiplication has always been a critical part of scientific and engineering applications. Recently, it has found extensive use in machine learning and other AI applications, which are growing in importance at all levels of computing. From the standpoint of high-performance systems, it will likely be a significant fraction of daily workloads. Therefore, even modest improvements in runtime or energy consumption can result in substantial gains in the overall performance of a typical high-performance system. Those gains may translate to meaningful reductions in the costs of operating such systems, especially those at the extreme. At the mobile levels, savings in power consumption may help yield longer battery life.

Kouya [1] noted that multiplication methods that process the matrices in blocks rather than whole rows and columns produced high cache hit ratios. One would expect those methods to result in significant memory reference locality. They are no better complexity-wise than their non-blocked counterparts. An energy perspective, however, may lend more value to the locality advantage. Unlike time efficiency, which relies on patterns of repetition of operations, energy efficiency is affected more by the use of physical assets. The two effects are not unrelated, of course. Therefore, block-oriented methods should show more energy efficiency since they rely on the SRAM-based cache, which is known to be energy efficient. Processing whole rows and columns relies more on the DRAM-based main memory, which is less efficient. Functional unit usage patterns should have similar effects. For example, in 1968, S. Winograd suggested a variation on the standard multiplication method [2] that ran faster. It was still cubic in complexity, but since it cut the

number of element multiplications in half, there may potentially be some energy benefits to expect. Also, depending on how algorithmic operations map to the hardware, the rate at which energy is consumed (power) may vary. Consuming energy at high power levels may cause equipment to experience failures due to high temperatures or may cause batteries to drain too fast. In those situations, power consumption may be more significant. In the context of this report, the reader should note that the terms energy and power, the time rate of energy, will not be used interchangeably. The authors will use the phrase *energy profile* or *behavior* to refer to all energy concerns in general, including power draw.

Strassen's divide–conquer multiplication [3] reduced the number of element multiplications for the 2×2 case to 7, down from 8 in the standard procedure. Winograd showed that result to be optimal for element multiplications and reduced the number of additions to 15, down from 18 in the original Strassen procedure, which Probert [4] proved optimal for additions for all Strassen-like methods. Strassen-like methods tend to be block-oriented since they are recursive, but they involve large memory overheads. A naive implementation may cause an excessive amount of memory allocation and motion. Strassen-like methods perform significantly fewer multiplications than the school pen–paper method at the expense of substantially more additions. Multiplications tend to be more expensive in terms of hardware and cost more energy. In addition, modern processors strive to offer multiplications that cost fewer cycles of latency to keep up with tight pipelines [5]. Aggressive timings may raise power consumption when energy costs are kept comparable. These variations in hardware and its usage should be interesting for energy concerns.

Based on the preceding ideas, this paper reports on an extensive experimental study of the energy and the power characteristics of six matrix multiplication algorithms. The algorithms are described fully in Section 3. Here is a summary.

1. A straight implementation of the standard/school algorithm;
2. Winograd's improvement on the standard method;
3. A naive implementation of Strassen's divide-and-conquer algorithm;
4. A naive Winograd's variant of Strassen implemented similarly to the previous one;
5. A memory-optimized Strassen that eliminates the allocation of temporary memory blocks used for calculations;
6. An algorithm similar to the previous one but with some memory motions eliminated.

The study focused on the case of square matrices crucial to most applications. It collected empirical energy performance and other data from an HPC-class AMD EPYC processor, including timings, cache misses, and memory bandwidth data. The algorithms varied in how they used the physical hardware. The study considered the following factors: functional unit usage, memory allocations, and memory motion (copying around regions in memory). It suggested memory allocation strategies to optimize energy behavior for the divide-and-conquer methods based on hardware usage. The choice of processor was significant since it is part of a family used in many high-performance systems (some on the https://top500.org/ (accessed on 24 March 2024) list). Thus, the findings should credibly approximate what to expect in those environments. This work attempts to elucidate the links between the usage patterns of energy-significant hardware and expected energy behaviors. It should, hopefully, provide valuable insights for future optimization efforts and may contribute to a deeper understanding of these computations. It may also help design algorithms that address energy concerns in better ways.

The organization of this paper is as follows. Section 2 presents a review of related work. Section 3 describes the characteristics of interest of the investigated algorithms. Section 4 describes methods and materials used in this research. Section 5 presents results and a thorough discussion of findings. Lastly, Section 6 offers some conclusions and proposes directions for future research.

2. A Literature Review

This section reviews a selection of relevant work in chronological order. Table 1 at the end of the section summarizes the main contributions for convenience.

Boyer et al. [6] focused on reducing the amount of memory used by Strassen–Winograd matrix multiplication. A highly interesting feature of the study was the development of several partially and fully in-place calculation schedules. Also, they developed a generic way to transform non-in-place algorithms to in-place ones with less overhead. They did not show that the memory-optimized versions ran faster (due to the drastic improvement in the utilization of the cache memory in many cases). The study did not consider stability or speed issues. It did include a cursory timing table, however.

Ren and Suda [7] studied power-efficient software for large matrix multiplication on multi-core CPU–GPU platforms. They measured and modeled the power of each component, considering frequency, voltage, and capacitance. They proposed a method to save power via multithreaded CPUs to control two parallel GPUs. They showed that their method saved 22% energy and sped up kernel time by 71%. However, their method depended on problem size and hardware configuration.

Ren and Suda [8] (second paper) optimized large matrix multiplication on multi-core and GPU platforms for power efficiency. They used block matrix partitioning, shared memory, and multithreading to improve the algorithm. Their enhanced kernel was 10.81 times faster than the original and saved 91% more energy. However, their power measurement, power model, and power-saving strategies may not be accurate or generalizable for other components, problems, or algorithms.

Yan et al. [9] compared parallel matrix multiplication algorithms on shared-memory machines in terms of performance, energy efficiency, and cache behavior. They studied loop chucking, recursive and hybrid tiling, and Strassen's algorithm. They found that the Intel MKL *cblas_dgemm* was the best for large matrices and small or medium threads. The hybrid tiling improved locality and performance. However, Strassen's algorithm had a lower cache hit rate and higher memory bandwidth usage. Their study lacked hardware features and software tools for power and energy profiling.

Legaux et al. [10] optimized matrix multiplication for multi-core systems with data access, vector unit, and parallelization techniques. They used Halstead metrics to evaluate software complexity and efficiency trade-offs. They showed that SSE and OpenMP versions were faster, but the SSE version was harder to develop. However, their method depended on architecture and had high development costs for some optimizations. Those factors limited their method's feasibility in some contexts.

Lai et al. [11] optimized the Strassen–Winograd algorithm for arbitrary matrix sizes on a GPU using techniques such as an empirical model, multi-kernel streaming, dynamic peeling, and two temporary matrices. They surpassed previous GPU implementations and showed Strassen's algorithm's practicality. However, they ignored the CPU–GPU communication cost, other fast algorithms, and Strassen's algorithm's numerical stability.

Kouya et al. [1] improved multiple precision matrix multiplications with Strassen's and Winograd's algorithms. They compared their speeds and accuracies for different precisions and block sizes. They used their technique for LU decomposition and checked the numerical properties on well-conditioned and ill-conditioned matrices. They showed that Winograd's was faster than Strassen's, and both recursive algorithms reduced computation time for LU decomposition. However, they lost significant digits for ill-conditioned matrices and required more tuning and parallelization for efficiency.

Sheikh et al. [12] developed a multi-objective evolutionary algorithm for parallel task scheduling on multi-core processors, optimizing performance, energy, and temperature together. Their approach beat existing techniques, as shown by experiments. The paper built on previous work on performance and temperature optimization.

Zhang and Gao [13] compared standard and Strassen tile-based matrix multiplication algorithms on high-density multi-GPU systems. They found that the standard method outperformed the Strassen method on these systems due to the performance gap between multiplication and addition operations. They concluded that the performance ratio of these operations determines the best matrix multiplication algorithm for new architectures.

However, their study was specific to the hardware configurations tested and may not apply to other systems or future technologies.

Kouya et al. [14] (second paper) compared parallelized Strassen and Winograd algorithms for multiple precision matrix multiplications using MPFR/GMP and QD libraries. They used thread-based parallelization to improve performance. They also showed that the Strassen and Winograd algorithms can reduce costs and increase speed in QD and DD precision environments.

Jakobs et al. [15] examined vectorization techniques' impact on dense matrix multiplication algorithms' power and energy. They used loop unrolling, frequency control, and compiler optimization levels. They showed that vectorization and loop unrolling reduced power up to 10% or more. The results varied by architecture and frequency. The study was limited to Intel's Sandy Bridge and Haswell architectures.

Muhammad and Islam [16] improved matrix–matrix multiplication performance on the .NET platform by loop restructuring of sequential and blocking algorithms. They found KIJ and IKJ reordering the best due to fewer cache faults and reads. However, they did not discuss the real-world implications or the applicability of these findings.

Haidar et al. [17] explored power usage control and its impact on scientific algorithms' performance. They used representative kernels and benchmarks and PAPI's *powercap* component for measurement and control to provide a framework for understanding and regulating power usage in scientific applications. They focused on identifying algorithms that benefit from power management strategies and gave observations and recommendations for energy efficiency in scientific algorithm development and execution.

Fahad et al. [18] measured on-chip power sensors and energy predictive models using PMCs against external power meters, finding significant errors in both methods. On-chip sensors had up to 73% average and 300% maximum error. Predictive models had up to 32% average and 100% maximum error. They also found that inaccurate energy measurements can cause up to 84% of losses in energy for dynamic energy optimization.

Oo and Chaikan [19] improved energy efficiency and performance of Strassen's algorithm for matrix multiplication on a shared memory architecture by using loop unrolling and AVX. They achieved 98% speedup and 95% savings in energy compared to the method without unrolling. However, their method depended on the loop unrolling factor, the code size, the number of registers, and the hardware configuration.

Kung et al. [20] addressed performance issues in bandwidth-constrained systems that are prevalent today. They proposed an algorithm to optimize memory bandwidth for a widely used block-oriented matrix multiplication. The goal was to find a balance between compute and memory bandwidths and tilted sharply against the memory access trends in the current state of technology. Their approach relied on shaping constant bandwidth blocks (self-contained units of computation that fit in local memory with fixed bandwidth needs to external ones). Adjusting block properties allowed the algorithm to control the ratio of (slow) external memory access to (fast) computation rate and overcome memory bottlenecks. Optimizing local memory (the cache) block bandwidth while limiting the less power-efficient DRAM-based external one should also provide energy benefits in addition to the reported performance boost, which makes such techniques of interest to our research.

Oo and Chaikan [21] optimized the Strassen–Winograd algorithm for arbitrary matrix sizes on a GPU using techniques such as empirical modeling, multi-kernel streaming, dynamic peeling, and two temporary matrices. They surpassed previous GPU implementations and showed Strassen's algorithm's practicality. However, they ignored CPU–GPU communication cost, other fast algorithms, and Strassen's algorithm's numerical stability.

Jammal et al. [22] compared the energy performance of a definition-based algorithm against two basic divide-and-conquer ones on an HPC-class Haswell Intel processor. They showed that the divide-and-conquer algorithms were the best in terms of power and energy consumption when the calculations fit in the cache. Both continued to be the most energy-efficient regardless of placement in memory. A notable result was that the definition-based

algorithm took the lead in power consumption, but not overall energy, for sizes that spilled over to the main memory. The authors described their results as preliminary.

Table 1. A summary of reviewed previous work.

Study	Year	Key Contribution
Boyer et al. [6]	2009	Developed memory-efficient scheduling for Strassen–Winograd's matrix multiplication, reducing overhead.
Ren and Suda [7]	2009	Optimized large matrix multiplication on multi-core and GPU platforms using CUDA, focusing on power efficiency.
Ren and Suda [8]	2009	Developed power-efficient software for large matrix multiplication on multi-core CPU–GPU platforms, measuring and modeling power components and proposing a method to save power via multi-threaded CPUs controlling two parallel GPUs.
Yan et al. [9]	2012	Compared multiple matrix multiplication parallel algorithms on shared memory machines for performance and efficiency.
Legaux et al. [10]	2012	Explored various optimizations for parallel matrix multiplication on multi-core systems, evaluating the trade-off between software complexity and computational efficiency.
Lai et al. [11]	2013	Optimized the Strassen–Winograd algorithm for matrices of arbitrary sizes, achieving significant speedups over existing methods.
Kouya et al. [1]	2014	Evaluated multiple precision matrix multiplications using parallelized Strassen and Winograd algorithms.
Sheikh et al. [12]	2015	Developed a multi-objective evolutionary algorithm for parallel task scheduling on multi-core processors, optimizing performance, energy, and temperature.
Zhang and Gao [13]	2015	Investigated performance of standard and Strassen matrix multiplication on high-density multi-GPU architectures.
Kouya et al. [14]	2016	Compared parallelized Strassen and Winograd algorithms for multiple precision matrix multiplications using MPFR/GMP and QD libraries, showing cost reduction and speed increase in QD and DD precision environments.
Jakobs et al. [15]	2016	Explored reduction in power and energy consumption for matrix multiplications through vectorization on modern computer systems.
Muhammad and Islam [16]	2017	Evaluated matrix multiplication performance in Virtual on the .NET platform, analyzing the performance of loop reordering and blocking algorithms in virtual machine environments.
Haidar et al. [17]	2019	Presented a framework for understanding and managing power usage in HPC systems, aiming to improve energy efficiency without compromising performance.
Fahad et al. [18]	2019	Compared methods for measuring the energy of computing, highlighting significant errors.
Oo and Chaikan [19]	2021	Studied effects of loop unrolling in an energy-efficient Strassen algorithm on shared memory architecture.
Kung et al. [20]	2021	Proposed an algorithm to optimize memory bandwidth for block-oriented matrix multiplication.
Oo and Chaikan [21]	2023	Enhanced Strassen's algorithm for power efficiency using AVX512 and OpenMP on multi-core architecture.
Jammal et al. [22]	2023	Conducted a preliminary empirical evaluation of the power efficiency of basic matrix multiplication algorithms.

3. The Algorithms

In classic complexity analysis, designers are concerned with the patterns of mapping the most frequent operations to time or memory. Efficiency depends on the growth of functions that express those patterns. The main idea in this work is that a careful mapping of such operations to the hardware is the key to designing power efficiency and, ultimately, total energy budgets. In some cases, the algorithms may offer opportunities for choosing an energy-efficient operation, such as dividing by two, which maps to highly efficient hardware. In others, there is a choice of circuits that serve the same purpose functionally but vary in energy behavior. A design focused on speed may result in a circuit that consumes a lot more energy or, worse in some cases, increases consumption rates (power) to maintain a time advantage. This section describes the investigated algorithms and highlights the differences in hardware usage patterns that are likely significant to their energy profiles. They vary mainly in how the frequent operations map to functional units, the local memory footprint, and the data motions in the memory. It should clarify the experimental design and perhaps help with interpretation of the outcomes. Table 2 presents a coding scheme for the algorithms and a reminder of their generic complexity. In each case, the complexity term is the exact number of element multiplications. For convenience, the codes will be used from this point on to refer to the algorithms.

Table 2. Algorithm coding scheme for reference purposes.

Code	Algorithm	Complexity
DEF	Standard definition-based	$O(n^3)$
WINOG	Winograd's improvement on the standard	
STRAS	Naive Strassen's divide–conquer	$O(n^{\log_2 7})$
WINOGV	Naive Winograd's divide–conquer variant	
STRAS2	Memory-optimized Strassen (*temps* in-place)	
STRAS3	STRAS2 + return product matrix in-place	

DEF is a straight implementation of the definition of matrix multiplication based on the inner product of rows into columns. It is simply the school pen–paper algorithm. WINOG is Winograd's improvement [2] on the standard, which cuts the number of element multiplications roughly in half without changing the base complexity. Both of these algorithms process whole rows and columns. Their performance should suffer relatively quickly as matrix sizes increase and cache utilization drops. The number of additions, however, increases 1.5 times in WINOG. So the net effect is not readily apparent, given that multiplication tends to be more energy expensive than addition.

The remaining algorithms are divide-and-conquer and are typically implemented recursively. A quick overview of the base method helps highlight the differences between them. These algorithms generally operate on $k \times k$ submatrix blocks (*k-blocks*, $k \leftarrow \frac{n}{2}, \frac{n}{4}, \ldots, 1$) of $2k \times 2k$ inputs and output on each recursive iteration, dubbed k to simplify the discussion. There are seven of these blocks included for the following iteration (therefore, the number of recursive iterations is $\sum_{i=0}^{k-1} 7^i = \frac{1}{6}(7^k - 1)$). These are combined to calculate and return an iteration's product matrix. Calculating these and holding the iteration input and output submatrices requires extra k-blocks. More are needed to return the iteration product matrix ($2k \times 2k$, i.e., four more k-blocks). Those blocks are common to all except STARS3 and thus are omitted from the discussion. It helps when designing these algorithms to distinguish temporary blocks (*temps*) used during calculations to hold expression partial and final results from those used to store iteration input and output submatrices (*submats*).

A careless implementation could use up to 37 blocks per iteration: 25 *temps* (18 sum and 7 product) and 12 in/out *submats*. With reasonable effort, one should be able to manage the block allocation to reduce its overhead considerably. Standard calculation sequences offer natural opportunities to reuse *temps*, which helps reduce the memory footprint, especially in local memory that is faster and more power efficient. For example, even a beginner

would quickly see that 8 of the sum *temps* may be reused to reduce the total allocation to 29 blocks. Also, in a straight implementation, an iteration k fills the k-blocks from its $2k \times 2k$ inputs and copies the resulting ones to its $2k \times 2k$ output. Filling those involves considerable back-and-forth copying.

STRAS follows a natural calculation schedule as suggested by a standard formulation (such as the one in [23]). It is considered naive for this reason. The natural flow allows the reuse of eight *temps* (as mentioned earlier) and four *submats*. Therefore, STRAS needs 17 *temps* and 8 *submats*, a total of 25 in iteration k. No memory allocation or motion optimizations were applied except an obvious allocation strategy: allocate blocks as late as possible and free blocks as early as possible, which is considered in this work to be the baseline for the family of methods. This allocation strategy needed 18 blocks at first, with four released early. STRAS retains four *temps* to calculate the outputs used to fill the iteration return matrix. Hence, its block allocation pattern was 18–14 and down to 4 before and after the recursive calls in each iteration—compared to the the 25 needed otherwise. WINOGV followed the same plan. It benefited from the recursive reduction in additions to hold slightly fewer temps. Together with the significantly fewer overall multiplications shared with STRAS, WINOGV should have better energy performance. Table 3 compares how the algorithms used memory.

Table 3. A summary of k-block usage per typical recursive call k. The last columns show an example of how much memory is held at most in iteration $k = 6$ for $n = 64$ to make a concrete sense of the differences between the algorithms.

Algorithm	Block Allocations		Block Motions	Max Holdings		Example	
	Max	Min		Blocks	Elements	Blocks	Elements
STRAS	18–14	4	12	$14(k-1)+4$	$14(4^k-1)/3+2$	74	19,112
WINOGV	16–14	4	12	$14(k-1)+2$	$14(4^k-1)/3-k$	72	19,104
STRAS2	12	4	12	$12(k-1)$	$4(4^k-4)$	60	16,368
STRAS3	8	-	8	$8(k-1)$	$8(4^k-4)/3$	40	10,912

STRAS2 is a memory-optimized multiplication technique that focuses on eliminating the *temps* and only keeping the *submats*. It employs the calculation schedule of Table 3 from [6] that uses the input *submats* as *temps* instead of creating extra ones. STRAS2 needs fewer blocks (good for caching) but keeps those that would have been freed earlier in STRAS until the schedule finishes (cached longer). The algorithm behind STRAS3, like STRAS2, does not create *temps*. It also eliminates the need for some *submats* by constructing the return product matrix in place, i.e., directly in the original $n \times n$ output matrix. It eliminates the need to copy from the output *submats* and thus reduces some of the memory motion. So it has the smallest memory footprint per recursive call, *and* it reduces the need to move data around memory. It should show the best energy performance of the lot.

Rather than a complex full implementation, STRAS3 simulates the power behavior. It fully implements the memory behavior of STARS2 in addition to in-place product matrix generation. However, instead of updating the row and column positions of the output block for each recursive iteration, it reuses the first position. The position calculation requires a constant time overhead per iteration. It should have a modest effect on the power draw relative to the memory access. Therefore, STRAS3 focuses on the parts that exercise the most relevant hardware for the energy rate. The rationale for the simulation is as follows: (a) expect a fairly representative power footprint and (b) inexpensively assess against the relatively cheap STRAS2. More importantly, it illustrates a point about power assessment. It should be reasonably accurate as long as the simulation exercises the same hardware in the same or close enough ways, which should help credibly check if more complex algorithms are worth it before committing to a full implementation.

Strassen-like algorithms rely on cutting the input size in half on each iteration, so they work naturally with input sizes that are powers of two, with no loss of generality since

one may pad the rows and columns of any matrix with zeros to change its dimensions without affecting its arithmetic value. The implementations in this work automatically upscale the dimensions to the nearest power of two (so 300, 400, and 500 are processed as 512). Therefore, some cases will be sparse, while others will be dense, which may be interesting from an energy perspective. Finally, the implementations do not halt recursion at an appropriately small block size and switch to a faster procedure, as is customary. They recurse down all the way to remain faithful to the method.

4. Materials and Methodology

The methodology used in this study was based on the established techniques for experimental measurement of energy and power on modern CPU platforms that were developed by some of the authors of this report and detailed in [22,24]. The main difference was the shift to another CPU platform. Fortunately, more modern CPUs, circa 2021, provided access to better profiling options. A summary of the methodology is as follows. To empirically determine the energy performance that one could credibly ascribe to the fundamental way a computation was configured, i.e., the algorithm: (a) the code is isolated on one processor core, and (b) as much influence of the runtime environment as possible, considered noise in this study, must be eliminated. Adverse influences could arise from the machine code representation of the computation, the OS threading model, and how the processor executes the code. In particular, power management mechanisms in modern CPUs can completely mask the natural energy behavior. Ultimately, the empirical evidence relies on high-resolution data from probes inside modern CPUs to dynamically control the run voltages and frequencies in response to loading conditions. Therefore, with a careful setup, there can be a credible argument that most of the measurements stem from the computation style. This backboxing approach eliminates the need for complex simulations that rely on detailed knowledge of the inner workings of the CPU and keeps the focus on the algorithm.

Timing data were collected as a control to ensure correct programming and expected asymptotic behavior. Experimentation showed that the readings from the profiling tools were consistent and close in value. An average of twenty to fifty sampling runs was sufficient to get a reliably convergent value for a single reading (to at least three fractional digits). For examples, see Table 4. These figures seem to confirm that the experimental setup was quite successful at eliminating noise from the runtime environment. Runs up to fifty times revealed little added value in the context of the experiment beyond guaranteeing convergence to four fractional digits. Therefore, readings for dimensions 3000 and 4000 were from an average of 20 runs due to their heavy time cost; the rest were from 40–50 runs.

Table 4. Sample STRAS power readings (watts) for matrix dimension 1100.

Average Power	Runs
12.91012346	7
12.91018765	10
12.91014681	20
12.91013579	30
12.91009531	40
12.91008022	50

The following describes the environment and procedures used in the experiments, the characteristics of the test datasets, the executable code, and the tools used to collect data.

4.1. Experimental Environment and Procedures

Table 5 lists the specifications of the testbed machine. The main feature is the AMD CPU from the third generation of the high-performance EPYC 7003 series (code name Milan). Cores are organized in blocks, called core complexes (CCXs), that combine 2–8 *Zen 3* cores with a sizable multilevel cache. A silicon die can have one or more CCXs (plus necessary off-core functions). The 7313 has four, with each on a separate die. This configuration should work well for thermal cross-noise affecting the core chosen to run the

experiments. Some experimentation revealed that readings were not affected by the system case being open or left closed or case fans on/off.

Table 5. The test machine: L1/L2 caches are private (L1 separate 32 KB instruction and data). Cores on the same CCX (core complex) share the L3. In the 7313, four cores vie for 32 MB of L3 cache. SMT is AMD's implementation of hardware threads.

Processor	**AMD EPYC 7313 (Milan) 16-Core (4×4 CCX) 32 SMT**		
Cache (SRAM)	**L1** 1 MB	$2 \times 16 \times 32$ KB	8-way
	L2 8 MB	16×512 KB	8-way
	L3 128 MB	4×32 MB	16-way
Main Memory (DRAM)	64 GB		
Operating System	Linux Ubuntu 22.04.3 LTS 64-bit		

The remainder of the section lists the measures and precautions put in place by the researchers to minimize environmental noise and ensure a reliable estimate of the energy profiles under investigation. These are related to the experiment environment and procedures, the CPU, or the operating system (OS).

1. Disabled the CPU cooler fan: prevent cooling the chip package based on different loading conditions, which allows the experimental loads to run under consistent thermals.
2. Configured a cool-off period between test runs: run scripts check the CPU temperature between runs and apply a time delay to return to a consistent baseline temperature of 30 °C before the following run could begin.
3. Disabled SMT: hardware-managed threads share a single physical CPU core in unpredictable ways from a computation viewpoint.
4. Disabled the power management components responsible for regulating the energy behavior of the CPU (core performance boost in AMD terminology). This function depends on hardware behaviors set by the CPU manufacturer. It could significantly alter energy patterns in unpredictable ways. Hence, it prevents an even comparison of the desired effects due to the test loads.
5. Set the *isolcpus* kernel boot parameter to explicitly restrict the OS process scheduler and load balancing algorithms from using a designated set of CPU cores reserved for the experiment (Cores 0–3 from the same CCX in this case).
6. Set the core affinity for the experiment's code via the profiling tool to ensure that the code ran exclusively on Core 0.
7. Set *perf_event_paranoid* in Linux to -1 (allow all events), as required by the profiler. The default settings usually restrict access to event monitoring for security reasons.
8. Disabled the NMI (non-maskable interrupt) watchdog on Linux (also recommended by the profiler) to stop periodic check overhead to improve the reported results.

4.2. Test Datasets

The datasets consisted of random square matrices ranging in dimension from 50 to 2500. Divide-and-conquer methods adjust the dimension to the nearest power of two. The range span ensured that the algorithms work across the three levels of cache well into the DRAM. Matrix elements were double-precision (8-byte) floating point numbers, the most common operand type in typical high-performance applications, based on randomizing the bit patterns rather than the stored values. There is some evidence [25] based on extensive experimental work that suggests power consumption depends on the data value being read or written. Randomizing stored bits should help eliminate any bias due to those effects.

The patterns used in the study only randomized the lower 52 bits that hold the fractional part of the number in the prevalent IEEE encoding. The upper bits, exponent and sign, were fixed so that values were in the interval $[2.0, 4.0)$. The rationale is as follows. Each application will likely work, for the most part, with a different range of values depending on the most frequent calculations and the application domain. The upper bits will not

likely change frequently, in general. The lower 52 bits, however, are guaranteed to change constantly throughout any computation in every application. Therefore, randomizing them will be a good representation of typical behavior involving floating point calculations. The actual range of values will not matter (any will do for this line of research). Figure 1 shows a sample of the random patterns used to fill test matrices. The datasets were pre-generated and saved in binary files accessible from any C/C++ program in order to be reusable. All algorithms used the same test data to keep things even.

Figure 1. Sample of test data simulating double-precision floating point calculations to illustrate bit patterns used for matrix elements and the corresponding numerical values.

4.3. Executables

The authors implemented the algorithms in standard C++. A multiply function calls code that implements an algorithm in a different source file. Therefore, separate executables for each case were generated via the GNU compiler collection (gcc 11.4.0) using the base command: gcc-lstdc++-lm. The experimental methodology calls for producing code as faithful as possible to the algorithms it purports to execute to be able to argue about the underlying methods. The machine code, in practice, is composed of instructions that reflect a combination of the hardware and the compiler. Optimizing compilers are concerned with efficient execution and can make the code bear little resemblance to the original computations described by an algorithm as long as the code runs correctly. Therefore, it may be desirable to turn off optimization. For this work, only the lowest level, the default, was allowed (compiler flag-O0). It performs basic optimizations, such as loop unrolling, which is arguably more faithful to the algorithms. Unrolling resembles hand applications of the algorithm (repeating based on a cheap decision) more than costly, hardware-dependent branches that machines go through. Thus, the resultant code should yield a realistic estimate of what users can expect while remaining reasonably faithful to the methods.

Higher optimization levels attempt to improve code performance at the expense of compilation speed, typically via aggressive code changes. For example, the next level in gcc (accessed via optimization flag-O2) performs common subexpression elimination [26]. It replaces expressions that result in the same values with variables for reuse if helpful. This optimization may alter the codes in different ways, making comparisons of behavior due to hardware use problematic. In general, such optimizations are counterproductive for this line of research as they may obscure the natural energy behaviors in question. It becomes harder to tell if the effects stem from the algorithms or from the compilation technology.

4.4. Profiling Tools

The study used the Linux version of the μProf profiler [27] version 4.0, which is provided by AMD and supports extensive power and thermal monitoring [28]. Some recent studies utilized the profiler for a variety of purposes. For example, Lane and Lobrano [29] used the profiler to analyze the memory bandwidth limitations in the second-generation EPYC chips. Lu et al. [30] used the profiler's energy reporting features to construct reliable thermal maps for an AMD Ryzen processor.

The main advantage of the tool for the study was that it provided a dependable source of nuanced core-specific metrics on the EPYC processor. It can report the runtime for a whole executable or break it down by function or thread. Moreover, it separates CPU time from user modules and system libraries. The detailed information helped confirm that core

and thread usage was consistent with expectations. It also revealed that most of the user module time (97–99% typically) was in the multiplication function. Therefore, the user module CPU time was an adequate choice since it captured the essential information while being more resilient to inaccuracies due to measurement resolution.

5. Results and Discussion

This section is divided into two main parts. The first presents the main results and discussions of findings about the observed energy behaviors. The second provides a detailed analysis of the memory trends behind those behaviors.

5.1. Main Results

Table 6 presents the main average power and energy consumption results (figures rounded to the first fractional digit). In the divide-and-conquer methods, lines indicate cases that share the same power of two matrix sizes, e.g., dimensions 300–500 upscaled to 512. These boundaries also mark very dense matrices for those methods. For brevity, runtimes will appear in a later figure set. Each power and timing data point is an average of multiple runs with bit-randomized datasets, as detailed in the previous section, to eliminate noise from the runtime environment. The power figures were based on spot averages reported by the profiler from the sensors for each run (a lightweight rapid sampling over the runtime interval [28]). The energy figures were calculated from average runtimes based on the times reported by the profiler and the average power figures. The table omits energy data for STRAS3 since it is a simulation of block behavior, which primarily affects power consumption. Its time and energy data will not be realistic and, therefore, not comparable to the other cases. The readings at dimension 50 (64 for the divide–conquer) seemed anomalous. The very short runtimes may interfere with the internal sampling process. It is safe to ignore that point as an outlier. For reliable instrumentation in tiny cases, readings should be from an average of repeated runs.

Table 6. Average power consumption in watts and overall energy in Joules (0.0 signifies tiny values of 3–18 mJ that were kept for context).

Dim.	Power (W)						Energy (J)				
	DEF	WINOG	STRAS	STRAS2	WINOGV	STRAS3	DEF	WINOG	STRAS	STRAS2	WINOGV
50	2.9	3.0	4.3	3.8	4.1	3.3	0.0	0.0	0.0	0.0	0.0
100	4.0	3.7	5.6	5.1	5.4	4.6	0.0	0.0	0.1	0.1	0.1
200	4.7	4.4	7.1	6.6	6.8	5.5	0.1	0.1	1.1	0.9	0.9
300	5.4	5.1	9.0	8.4	8.6	7.1	0.5	0.3	10.1	8.1	9.3
400	6.2	5.8	9.0	8.4	8.6	7.1	1.3	0.9	10.2	8.2	9.3
500	7.2	6.8	9.0	8.4	8.7	7.1	3.0	2.0	10.3	8.2	9.5
600	8.0	7.6	10.1	9.1	9.7	7.8	5.7	4.0	77.6	61.4	72.0
700	9.3	8.8	10.5	9.5	10.1	8.2	10.6	7.4	80.7	64.6	75.2
800	10.2	9.6	11.0	10.0	10.7	8.8	17.3	12.1	85.2	68.5	79.4
900	11.2	10.6	11.5	10.4	11.0	9.2	27.1	19.1	89.5	71.4	82.2
1000	12.4	11.7	12.0	10.9	11.4	9.7	40.9	28.8	93.9	75.1	85.0
1100	13.6	12.8	12.9	11.8	12.4	9.9	59.9	41.9	694.9	562.0	647.4
1200	15.0	14.1	13.2	12.1	12.7	10.2	85.3	60.1	714.7	576.3	662.6
1300	16.5	15.5	13.4	12.3	12.9	10.4	120.0	84.0	728.0	588.8	675.0
1400	18.1	17.1	13.8	12.7	13.3	10.6	165.7	115.2	748.4	607.9	695.7
1500	19.9	18.8	14.4	13.3	13.9	11.1	224.1	156.3	783.0	636.9	729.2
1600	21.9	20.7	15.1	14.0	14.5	11.7	301.6	210.8	820.5	671.8	768.2
1700	23.0	21.7	15.7	14.6	15.1	12.1	387.7	271.5	854.1	700.8	799.7
1800	23.8	22.5	16.4	15.3	15.8	12.7	493.0	342.6	892.7	737.0	836.8
1900	24.4	23.1	17.6	16.5	16.9	13.6	611.3	432.1	957.6	795.6	898.6
2000	24.9	23.5	20.5	19.3	19.7	15.8	757.7	524.4	1119.1	933.7	1047.0
2100	25.3	23.9	21.8	20.3	20.9	16.8	918.3	630.5	8251.4	6817.0	7657.8
2200	25.7	24.3	22.6	21.1	21.7	17.4	1113.9	751.2	8567.4	7090.4	7957.4
2300	26.1	24.7	23.6	22.1	22.7	18.2	1310.8	906.5	8953.0	7440.7	8319.2
2400	26.6	25.1	24.5	23.0	23.6	18.9	1527.6	1045.3	9290.9	7736.7	8651.7
2500	26.8	25.3	25.9	24.3	24.9	20.0	1780.3	1193.2	9833.8	8205.6	9133.0
3000	30.9	29.2	28.0	26.1	26.9	21.5	3631.4	2574.6	10,666.6	8751.6	9828.6
4000	38.3	36.1	36.1	33.8	34.7	27.9	10,107.7	7858.1	13,797.8	11,376.4	12,723.2

The table immediately reveals the advantage of small dimensions, up to 100, for the non-recursive methods (128 for the others). However, the differences in power draw or runtimes, and hence, energy consumption, were insignificant at those points, as expected. Figure 2 comparatively shows how energy seemed to track the runtimes, suggesting similar asymptotic trends. The asymptotic advantage of divide-and-conquer should only show for large enough matrices on a particular CPU. A closer look at timing data showed that the ratio of runtimes for STRAS to DEF shrank from up to 12 times slower to 1.4 at 4000. Therefore, for this CPU, the asymptotic advantage should start to show for larger sizes not too far from 4000. Moreover, the mildly memory-optimized STRAS2 was an average of 14% faster than the basic STRAS, a result somewhat consistent with the results in [6] (for the small dimensions reported there).

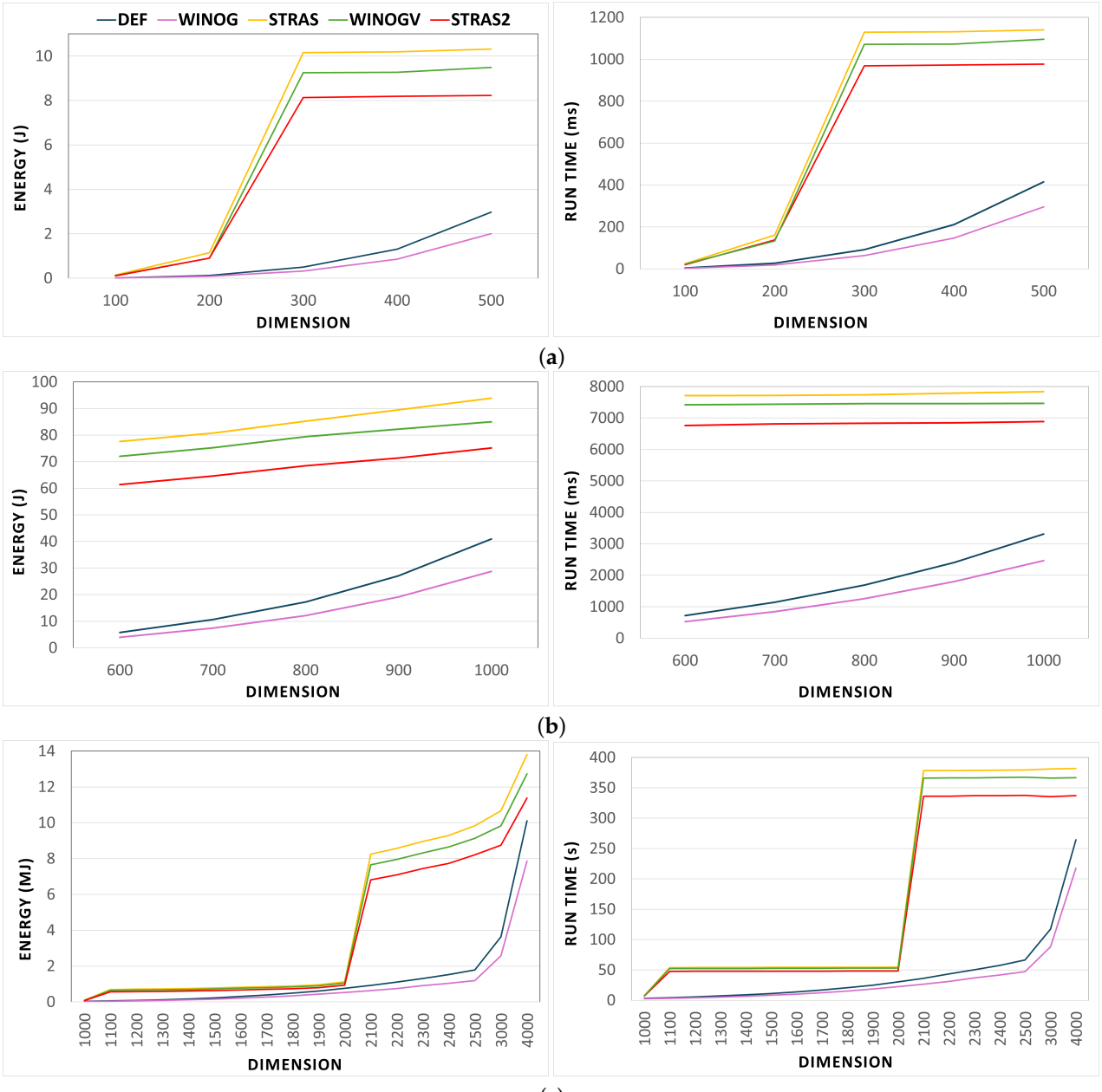

Figure 2. Total energy (**left**) and runtimes (**right**). The sharp turn at 3000 is due to missing data points. (**a**) Small matrix size lower range (inside 10 J). (**b**) Small matrix size upper range (inside 100 J). (**c**) Large matrices (up to 14K joules, likely DRAM).

The following parts separately discuss energy characteristics, analyze power performance, and close with some notes.

5.1.1. Energy Consumption

Energy trends in Figure 2 show DEF and WINOG to be drastically more energy efficient for most of the investigated sizes. Their advantages diminish significantly, however, at the larger end of the range. This should be expected based on the heavy costs of recursion incurred by the divide-and-conquer methods, which are hard to justify at the lower part of the range. As the matrix size rises, the runtime asymptotic advantage overcomes recursion overheads, translating eventually into energy consumption advantages. The energy consumption of WINOG was consistently better than that of plain DEF at an average of 42% (only 35% faster on average, though), with a widening gap as matrix size increased. Better utilization of functional units may explain that result. The readers may recall that WINOG performs half the multiplications and 1.5 times more additions than DEF. By shifting the computational burden to the less expensive hardware, WINOG can run more energy efficiently.

Among the divide–conquer methods, STRAS2 consumed the least amount of overall energy. In particular, it consumed 24% on average less than the basic STRAS. This is a significant improvement given the small effort taken to optimize memory usage. STRAS2 moves around significantly fewer in-flight memory blocks than the other two methods, which may explain its efficiency. The Winograd variant of the original Stassen, WINOGV, was also clearly better than STARS. The slightly faster WINOGV uses energy-significant hardware less by performing fewer additions. The payoff in energy was even better than that in time. In all cases, a faster runtime may explain some of the improvements in overall energy efficiency, but the bigger payoff in energy may be due to the more optimal use of energy-significant hardware or, perhaps, points to better consumption rates.

5.1.2. Power Performance Analysis

Power consumption findings were somewhat more interesting. Power consumption is critical for applications for which energy concerns are not just about total consumption. The rate of energy expenditure is crucial in mobile and high-performance applications. Optimizing for low power could lead to smaller, lighter batteries or to fitting heavier workloads within a power budget. Figure 3 shows the average power consumption for the divide-and-conquer group. Figure 4 shows the power for DEF and WINOG superimposed on the best two from the divide-and-conquer set for comparison.

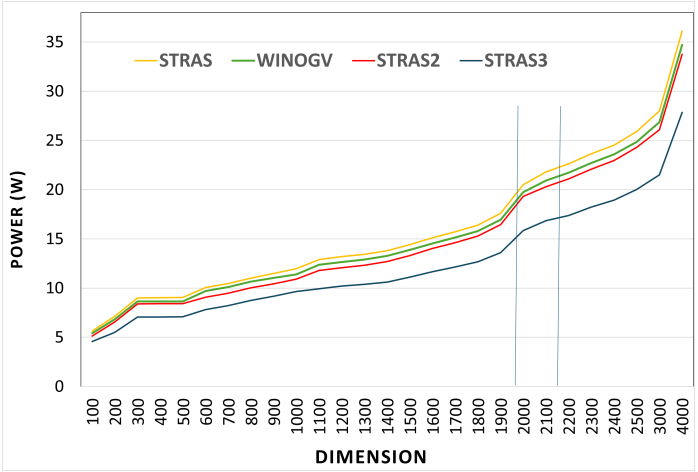

Figure 3. Average power consumption in watts for the divide-and-conquer algorithms.

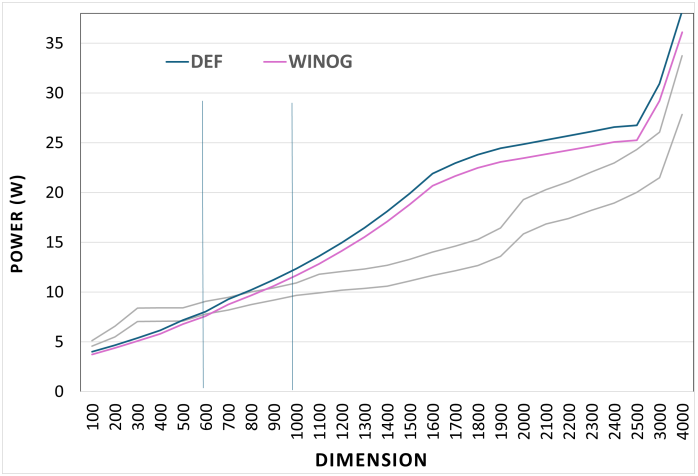

Figure 4. Comparative average power with DEF and WINOG (only STRAS2 and STRAS3 shown).

It is clear from Figure 3 that the memory-optimized algorithms, STRAS2 and STRAS3, outperformed their unoptimized counterparts throughout the range. The significant reductions in the memory footprint in both cases (more in STRAS3) may explain this result. In particular, the power draw was an average of 8.3% less in the mildly memory-optimized STRAS2 than in the naive STRAS (the baseline for the method). This may not seem like much, but the savings could add up for a system-wide workload for which multiplication (at various matrix dimension points) is a significant fraction. WINOGV was also consistently a little better than STRAS, likely due to its slightly reduced memory footprint (see Table 3) and clever use of functional units from a power viewpoint.

STRAS3 was better than STRAS2 since it implemented more aggressive in-place memory block management. The main difference between the two was that STRAS3 reduced memory motion also. The improvement due to STRAS3 was significant. Although STARS3 was just a simulation, it did include a valid implementation of most block behaviors relevant to power. A caution, however, is in order. Some of the cache traffic in STRAS3 will go unaccounted for due to writing the product blocks in the same position in the DRAM-resident $n \times n$ output matrix. That may exaggerate its performance, but the effects should be slight since those DRAM blocks are relatively few, and all the required block allocations still occur. Therefore, the STRAS3 results should at least indicate the potential for its added optimization. In-place algorithms tend to be more complex, with typically little advantage with regard to runtime. So they tend not to be attractive unless there are crucial limitations on storage. But in the case of STRAS3, the added complexity of the in-place calculations in the output seems to be well justified from a power consumption view.

Figure 4 shows WINOG to be consistently more power efficient than DEF (6% on average), as expected given the relatively reduced use of the typically power-hungry multiplication hardware. This edge may also explain why the payoff in overall energy consumption was greater than that in the runtime, as observed earlier. Moreover, DEF and WINOG were more power efficient than the best of the divide-and-conquer group through the upper range of small matrix sizes (dimensions 500–1000), after which the trend seemed to reverse. The region over which the reversal happened corresponds to a point where the power of two matrix sizes shifts from 512 to 1024. After that inflection point, all the divide-and-conquer algorithms consumed less power. The more efficient of the group overtakes earlier after that point. From dimension 1100 forward, the divide-and-conquer method seems to have a clear power advantage. In that range (1100–4000), STRAS2 was a significant 24% on average more power efficient than WINOG. It is also evident from the performance of STRAS3 that more aggressive memory optimization may be expected to lead to even better power efficiency gains.

A closer look at the power figures of the divide-and-conquer algorithms shows that the power gradually increases (converges) for dimension points that share the same power-

of-two matrix size as the matrix becomes dense. This sparse vs. dense matrix effect seems to support the idea that processing bit patterns dominated by zeros may consume less power. It also suggests that the dimension points corresponding to the dense cases (marked by lines) best depict the performance of the divide-and-conquer method. Therefore, these points should provide the most realistic comparison to the other two methods (see Table 7). Also, this trend encourages adding checks to cut recursions that will not contribute to non-zero elements in the product matrix to improve the power performance for the sparse cases even more. The energy consumption would also drop due to reducing the power and the runtime together.

Table 7. The relative advantage in power draw in the dense matrix cases (95.4% non-zero elements) for the pair of memory-optimized divide-and-conquer compared to WINOG. Negative percentages signify improvement. STRAS3 hints at the potential for optimizing memory motion.

Dimension	Power (W)			% Advantage	
	WINOG	STRAS2	STRAS3	STRAS2	STRAS3
500	6.8	8.4	7.1	24	5
1000	11.7	10.9	9.7	−7	−17
2000	23.5	19.3	15.8	−18	−33
3000	29.2	26.1	21.5	−11	−26
4000	36.1	33.8	27.9	−7	−23

5.1.3. Closing Notes

It is worth noting that the reader should treat the results discussed in this section as a worst-case for the divide-and-conquer algorithms. Practical implementations should stop the recursion at appropriate points and switch to some faster non-recursive procedure to avoid the overhead of excessive recursions, which should widen the advantage of those methods. From the viewpoint of energy behavior, that point should be when the power advantage starts to shift in favor of methods based on the row and column inner products (DEF or WINOG). For the testbed CPU, the data suggest that at dimension 1024, perhaps the shift should also close the runtime and total energy gap somewhat, which brings the asymptotic advantage of the divide-and-conquer methods to even smaller matrix sizes.

In [22], the authors commented on how much power these seemingly fundamental computations consumed on an HPC-class Intel Haswell CPU from 2014: as much as 80 watts at matrix size 1500. The AMD EPYC, a later CPU (2021), consumed 20 and 14 watts for DEF and STRAS, respectively, for the same size matrix. This is a statement on how costly the computation is and how far power technology has come in less than a decade.

5.2. Cache Miss Analysis and Other Memory Trends

A computation interacts with memory in different ways as it moves through the levels of a memory hierarchy. Cache misses significantly influence power and energy consumption [31]. The cache, typically in SRAM technology, is known to be much more power efficient than DRAM-based main memory. Therefore, from a power standpoint, the most significant event occurs when a computation spills into the main memory. Moreover, all cache levels tend to be on-chip nowadays, with relatively little differences in timing compared to the DRAM. Hence, it is more about differences in the general behavior of the algorithm when the computation is small enough to fit in the uppermost parts of the cache. As its size grows, it becomes more about success in managing the locality, with profound effects on speed and power. Behavior in the necessarily tiny Level 1 (L1) cache is the least interesting due to its inherently poor locality and the minor differences in timing and energy behavior. Therefore, the discussion in this section will lump L1 with the second cache level. In addition, the divide-and-conquer strategy involves breaking down larger matrices into smaller blocks that memory-intensive recursive functions calls processes. This approach can lead to premature cache spillover as data and stack frames vie for cache

space. This effect is crucial for cache efficiency, with consequent impacts on the energy performance of those algorithms in general.

With the preceding points in mind, Table 8 summarizes cache miss data collected for the L2 and L3 caches. The profiling tool did not provide straightforward cache miss counts per run. Instead, the tool provided access and miss counts per thousand instructions (PTI), i.e., normalized rates. Miss ratio PTIs for L2 and L3 caches may be calculated from those count rates. Absolute counts show pronounced jumps, which helps spot likely spillover points. Rates, however, tend to even out differences, leading to more subtle behavior. In addition, the profiler performs internal sampling and can provide more sample points for readings depending on runtimes. Longer runtimes allow for more samples and, hence, more accurate gauging. Therefore, results for larger dimensions involving more samples tend to provide more reliable figures than smaller ones. The table shows averages of ten runs. Two regions of interest inferred from the energy and power data are highlighted for closer examination.

Table 8. Average cache miss ratios per thousand instructions (PTI), the L3 figures are percentages. Regions of interest are shaded for highlighting.

Dim.	DEF		WINOG		STRAS		STRAS2		WINOGV		STRAS3	
	L2	L3	L2	L3	L2	L3	L2	L3	L2	L3	L2	L3
50	0.087	7.5	0.055	6.5	0.039	12.0	0.034	11.7	0.015	8.5	0.010	7.7
100	0.103	11.1	0.060	7.1	0.061	15.4	0.060	14.1	0.019	10.0	0.026	7.8
200	0.120	18.8	0.150	12.4	0.130	15.6	0.096	15.4	0.023	11.5	0.042	8.3
300	0.130	19.4	0.160	15.7	0.140	16.9	0.107	18.8	0.138	12.1	0.089	10.4
400	0.160	19.7	0.170	16.1	0.170	18.0	0.158	18.9	0.144	12.4	0.130	11.7
500	0.170	20.2	0.180	19.6	0.176	19.2	0.160	19.0	0.157	12.7	0.147	12.1
600	0.179	22.5	0.192	21.1	0.196	20.0	0.196	19.2	0.149	13.0	0.158	12.1
700	0.192	22.9	0.196	21.6	0.217	20.6	0.223	19.7	0.151	16.0	0.180	13.9
800	0.210	24.4	0.205	21.7	0.254	22.4	0.235	20.9	0.191	16.4	0.174	14.4
900	0.219	24.7	0.217	21.8	0.254	22.7	0.254	20.8	0.204	17.5	0.183	16.6
1000	0.230	25.4	0.217	22.1	0.257	22.7	0.256	21.2	0.268	21.0	0.298	18.9
1100	0.242	25.4	0.220	23.7	0.266	23.4	0.266	23.0	0.375	22.1	0.302	19.2
1200	0.242	25.7	0.220	25.4	0.267	24.4	0.281	23.3	0.380	22.2	0.313	19.6
1300	0.294	26.0	0.235	25.8	0.297	24.4	0.293	23.7	0.427	22.2	0.318	19.7
1400	0.303	26.0	0.235	26.1	0.331	24.5	0.305	24.3	0.429	22.7	0.329	20.3
1500	0.363	27.5	0.250	26.2	0.331	24.7	0.326	25.0	0.430	22.7	0.370	20.3
1600	0.370	28.3	0.258	26.9	0.343	25.2	0.337	25.3	0.430	23.0	0.427	20.3
1700	0.381	29.2	0.260	27.0	0.353	26.1	0.348	25.5	0.433	23.6	0.437	21.1
1800	0.397	30.6	0.267	27.0	0.351	26.7	0.351	26.1	0.435	24.1	0.440	22.0
1900	0.400	30.6	0.273	27.1	0.405	28.1	0.362	26.5	0.439	24.6	0.447	23.0
2000	0.414	32.1	0.280	31.6	0.428	28.3	0.402	27.2	0.440	25.1	0.453	25.1
2100	0.422	36.3	0.331	34.5	0.434	36.0	0.424	32.5	0.450	32.5	0.450	28.7
2200	0.436	38.8	0.361	37.2	0.459	38.1	0.435	34.9	0.454	32.9	0.460	28.9
2300	0.476	39.2	0.383	38.5	0.463	39.2	0.455	35.1	0.455	36.5	0.500	29.8
2400	0.490	41.3	0.450	40.0	0.493	39.5	0.480	36.5	0.470	37.4	0.576	29.8
2500	0.620	41.8	0.455	40.1	0.542	40.1	0.520	38.8	0.489	37.7	0.590	30.7

An examination of the L3 misses for the upper part of the investigated range in Figure 5 shows distinct surges around dimension 2000 (slightly later for STRAS3). Data points at 2000 correspond to dense matrices of size 2048 for the divide-and-conquer methods (sparse 3072 for STRAS3 at data point 2100). DEF and WINOG displayed similar behavior around dimension 1900. As a reminder, the data suggested that the memory-optimized divide-and-conquer showed 18–33% less power draw at the 2000 point (Table 7) relative to WINOG. The figure seems to support a probable range of spillover to DRAM between dimensions 1900 and 2100, as suggested by the energy and, to some extent, power data. Memory bandwidth data in Figure 6 also show a significant increase in memory activity in the probable region where the boundary between SRAM and DRAM may reasonably be estimated to be.

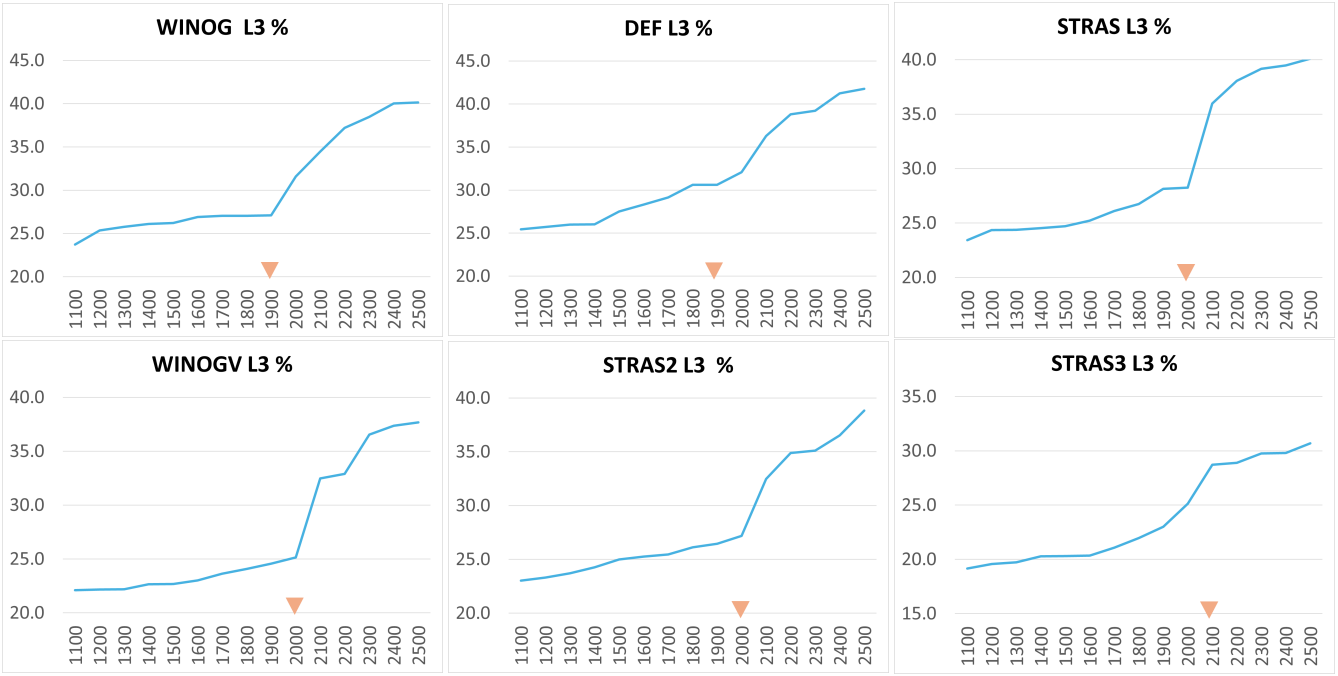

Figure 5. Level 3 average cache miss ratios (PTI) reported as percentages.

The slightly later spillover of the block-oriented divide-and-conquer group may stem from their superior locality management, which led to better utilization of the power-efficient cache. The energy data, however, suggest that those methods experience a surge in energy consumption initially, probably due to a runaway runtime overhead of recursion. Eventually, the asymptotic time advantage catches up at sufficiently large matrices and, with superior power, yields better energy performance. Optimizing memory may bring that advantage to smaller dimensions.

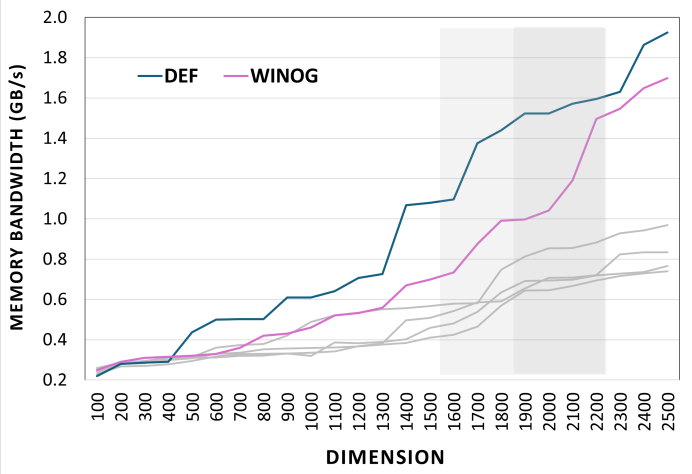

Figure 6. Average read/write main memory bandwidth for DEF and WINOG superimposed on the divide–conquer for comparison.

Furthermore, Figure 7 shows that the memory-optimized variants had better memory performance. Better locality management is probably behind that edge in power and the ensuing energy savings in STRAS2 and STRAS3 compared to STRAS and WINGV. Conversely, poor cache performance may be behind the much worse surges in memory activity that seem to occur significantly earlier in DEF and WINOG in Figure 6, which may account for the degraded power performance for the same range (Figure 4).

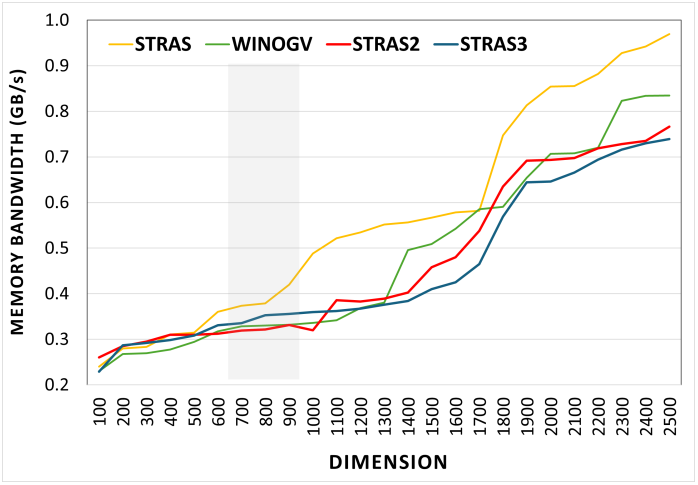

Figure 7. Average read/write main memory bandwidth for the divide-and-conquer group.

L2 cache trends, in Figure 8, proved to be more subtle but still discernible. As expected, the divide-and-conquer methods tended to miss significantly out of L2 earlier at around dimensions 700–900 compared to the other two at 1100–1200, with the memory-optimized STRAS2 and STRAS3 lingering a little longer (closer to the 900 data point). The 700–1000 range corresponded to processing matrices of dimension 1024 with an upshift to 2048 for the 1100 and 1200 points. This behavior corresponded to the second region of interest identified in the power curve from Figure 4, where power efficiency was turning away from DEF and WINOG. It seems to indicate that the computation size advantage of DEF and WINOG was no longer yielding power savings. The superior block management of the divide-and-conquer was starting to take over, perhaps. A glance at the probable region of spillover from L2 to L3 in Figure 7 reveals that it was likely where STRAS and WINOGV started to be less competitive with the memory-optimized variants in terms of memory bandwidth. It is also where WINOG is superior to DEF but starts to lose its power efficiency edge against the least-efficient divide-and-conquer methods, as Figure 9 shows.

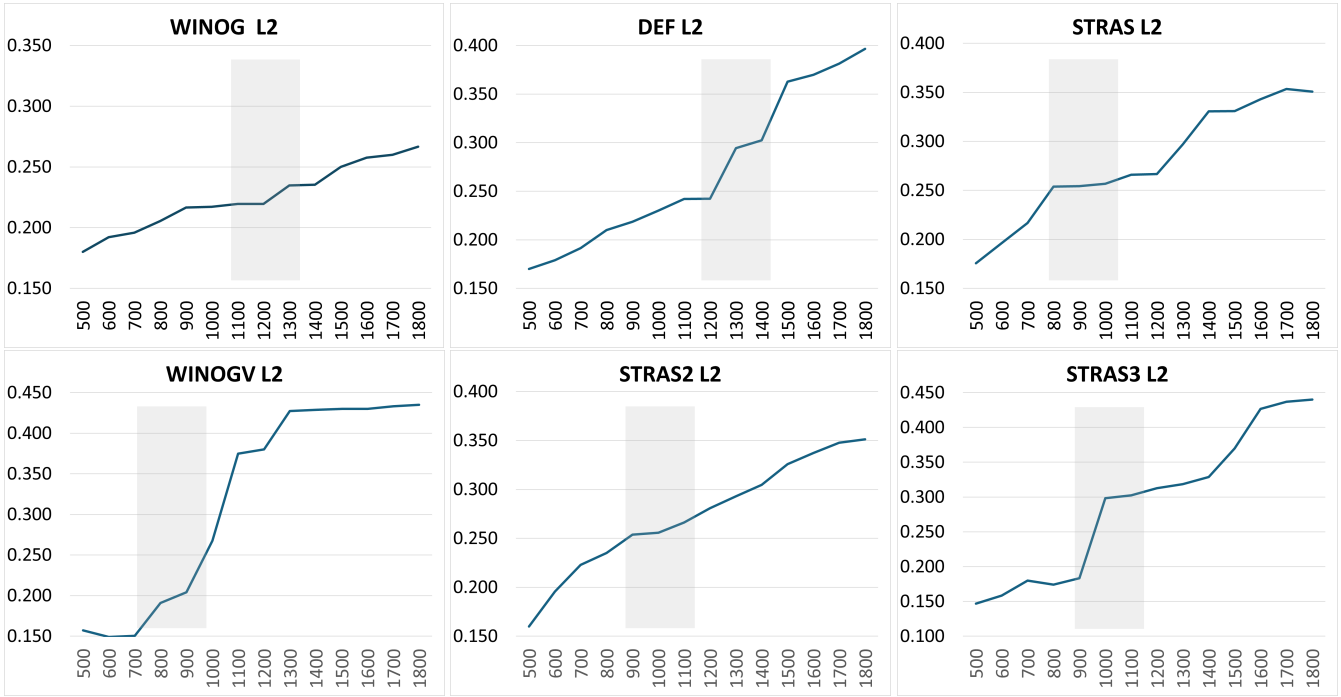

Figure 8. Level 2 cache miss ratios (PTI) with the probable spillover region to the L3 cache indicated.

Figure 9. Detailed view of the main memory bandwidth showing the probable L2–L3 spillover range.

To conclude this discussion, the authors note that the observed fluctuations in memory behavior data perhaps highlight the complex interplay between the algorithms and the memory subsystem. Nevertheless, the memory interactions did offer the insights needed to explain the observed energy behaviors. More notably, they supported expectations from prior knowledge of how the algorithms worked and how that was supposed to impact those behaviors.

6. Conclusions and Future Work

This study investigated the energy behavior of a selection of standard matrix multiplication algorithms for advantages, in terms of their total energy and power draw budgets, that one may attribute to their design and hardware use patterns. It also examined cache misses to make sense of consumption patterns and to determine the effects of interactions with the memory.

The immediate conclusion was that, at least from a power viewpoint, the Winograd variant of Strassen's original algorithm (WINOGV) should be the baseline for the divide-and-conquer method. The baseline should include at least in-place temporary block scheduling like STRAS2, which involves minimal effort. Also, switching to a fast, non-recursive procedure should be a must for the best effect. Either DEF or WINOG would do for such a procedure. This baseline implementation should yield slightly better performance than that reported in this paper for STRAS2. Timing and power data suggest it will not suffer computationally but could be significantly more power efficient (see data points 1000–1100).

However, for the best power and energy performance, a high-performance algorithm for large matrix sizes should go for the full extent of memory optimizations described in [6] on top of WINOGV. It should perhaps detect recursions that yield zero blocks in sufficiently large sparse cases. WINOG is preferable to DEF here for a fast procedure for appropriately small dimensions. The choice of the point at which the algorithm switches to that procedure should be custom-optimized for a system. That is not as open-ended as it may sound. The state-of-the-art in technology and the tight design trade-offs between hit times vs. miss rates often lead to similar configurations within a generation of processors, which likely lead to the same ends. The complexity of such an undertaking could be offset by including it in a system-wide library for frequent reuse. A multicore parallel implementation should probably also start from the previous guidelines.

The study also found the usage patterns of energy-significant functional and memory units to be reasonably good predictors of what to expect in terms of power and energy behavior. Improving energy budgets through optimizing the runtime under the same power draw is perhaps expected. However, optimizing hardware usage was also found to improve the power performance. For some applications, lower power budgets may be more critical. Reducing those budgets, in turn, also boosts energy gains as well. The power

advantage may be more valuable for a ubiquitous computation that is likely to make up a significant fraction of a large workload on a high-performance system where the savings may accumulate. Classical algorithm design keeps an eye on the growth of a dominant term of a function that expresses the relation of the repetitions of the most frequent operations with the input size. Similarly, an analysis based on the most frequently used hardware-significant units could guide the design of algorithms with better energy profiles. Moreover, this could lead to rethinking the design of those units to increase the savings.

The findings from the simulated case using STRAS3 seem to encourage an examination of a full implementation to better assess its advantage in a more realistic setting, especially in terms of power consumption. In particular, a study should perhaps focus on in-place calculations that minimize memory motion. Moreover, the SIMD architecture of GPUs makes them a natural fit for matrix multiplication. These platforms are now primary environments for applications that rely on that vital computation. It would be interesting to replicate the study on a popular GPU setting like those from Nvidia or AMD. It would also be interesting to formalize a framework for power complexity like that for time and space. Additional studies along the suggested lines may lend further support for seeking the efficiency inherent in algorithms by design. In particular, this may be useful for designing algorithms that manage the use of hardware with a critical influence on energy behaviors.

Author Contributions: Conceptualization, M.A.-H. and S.A.; methodology, S.A. and M.A.-H.; software, M.A.-H.; validation, S.A. and M.A.-H.; formal analysis, M.A.-H.; investigation, S.A.; resources, M.A.-H.; data curation, S.A.; writing—original draft preparation, M.A.-H. and S.A.; writing—review and editing, M.A.-H.; visualization, M.A.-H. and S.A.; supervision, M.A.-H.; project administration, S.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

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

Abbreviations

The following abbreviations are used in this manuscript:

HPC	High-performance computing
SMT	Simultaneous multithreading
PTI	Per thousand instruction
DEF	Standard definition-based [matrix multiplication]
WINOG	Winograd's improvement on the standard
k-Block	$k \times k$ submatrix block
STRAS	Naive Strassen's divide–conquer
WINOGV	Naive Winograd's divide–conquer variant
STRAS2	Memory-optimized Strassen (in-place *temps* calculation schedule)
STRAS3	STRAS2 + return product matrix in-place

References

1. Kouya, T. Accelerated multiple precision matrix multiplication using Strassen's algorithm and Winograd's variant. *JSIAM Lett.* **2014**, *6*, 81–84. [CrossRef]
2. Winograd, S. A New Algorithm for Inner Product. *IEEE Trans. Comput.* **1968**, *C-17*, 693–694. [CrossRef]
3. Strassen, V. Gaussian elimination is not optimal. *Numer. Math.* **1969**, *13*, 354–356. [CrossRef]
4. Probert, R.L. On the Additive Complexity of Matrix Multiplication. *SIAM J. Comput.* **1976**, *5*, 187–203. [CrossRef]
5. Fog, A. *Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs*; Technical Report; Technical University of Denmark: Kongens Lyngby, Denmark, 2022. Available online: https://www.agner.org/optimize/instruction_tables.pdf (accessed on 14 April 2024).
6. Boyer, B.; Dumas, J.G.; Pernet, C.; Zhou, W. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, Seoul, Republic of Korea, 28–31 July 2009; pp. 55–62.

7. Ren, D.; Suda, R. Power efficient large matrices multiplication by load scheduling on multi-core and GPU platform with CUDA. In Proceedings of the 2009 International Conference on Computational Science and Engineering, Vancouver, BC, Canada, 29–31 August 2009; IEEE: New York, NY, USA, 2009; Volume 1, pp. 424–429.
8. Ren, D.Q.; Suda, R. Modeling and optimizing the power performance of large matrices multiplication on multi-core and GPU platform with CUDA. In Proceedings of the International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland, 13–16 September 2009; Springer: Berlin/Heidelberg, Germany, 2009; pp. 421–428.
9. Yan, Y.; Kemp, J.; Tian, X.; Malik, A.M.; Chapman, B. Performance and power characteristics of matrix multiplication algorithms on multicore and shared memory machines. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; IEEE: New York, NY, USA, 2012; pp. 626–632.
10. Legaux, J.; Jubertie, S.; Loulergue, F. Experiments in parallel matrix multiplication on multi-core systems. In Proceedings of the Algorithms and Architectures for Parallel Processing: 12th International Conference, ICA3PP 2012, Fukuoka, Japan, 4–7 September 2012; Proceedings, Part I 12; Springer: Berlin/Heidelberg, Germany, 2012; pp. 362–376.
11. Lai, P.W.; Arafat, H.; Elango, V.; Sadayappan, P. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In Proceedings of the 20th Annual International Conference on High Performance Computing, Pune, India, 18–22 December 2013; IEEE: New York, NY, USA, 2013; pp. 139–148.
12. Sheikh, H.F.; Ahmad, I.; Fan, D. An evolutionary technique for performance-energy-temperature optimized scheduling of parallel tasks on multi-core processors. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 668–681. [CrossRef]
13. Zhang, P.; Gao, Y. Matrix multiplication on high-density multi-GPU architectures: Theoretical and experimental investigations. In Proceedings of the High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, 12–16 July 2015; Proceedings 30; Springer: Berlin/Heidelberg, Germany, 2015; pp. 17–30.
14. Kouya, T. Performance evaluation of multiple precision matrix multiplications using parallelized Strassen and Winograd algorithms. *JSIAM Lett.* **2016**, *8*, 21–24. [CrossRef]
15. Jakobs, T.; Hofmann, M.; Rünger, G. Reducing the power consumption of matrix multiplications by vectorization. In Proceedings of the 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), Paris, France, 24–26 August 2016; IEEE: New York, NY, USA, 2016; pp. 213–220.
16. Muhammad, A.; Islam, M.A. Performance evaluation of matrix multiplication in Virtual Machine. In Proceedings of the 2017 International Conference on Communication, Computing and Digital Systems (C-CODE), Islamabad, Pakistan, 8–9 March 2017; IEEE: New York, NY, USA, 2017; pp. 205–210.
17. Haidar, A.; Jagode, H.; Vaccaro, P.; YarKhan, A.; Tomov, S.; Dongarra, J. Investigating power capping toward energy-efficient scientific applications. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e4485. [CrossRef]
18. Fahad, M.; Shahid, A.; Manumachu, R.R.; Lastovetsky, A. A comparative study of methods for measurement of energy of computing. *Energies* **2019**, *12*, 2204. [CrossRef]
19. Oo, N.Z.; Chaikan, P. The Effect of Loop Unrolling in Energy Efficient Strassen's Algorithm on Shared Memory Architecture. In Proceedings of the 2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), Jeju, Republic of Korea, 27–30 June 2021; IEEE: New York, NY, USA, 2021; pp. 1–4.
20. Kung, H.T.; Natesh, V.; Sabot, A. CAKE: Matrix Multiplication Using Constant-Bandwidth Blocks. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MI, USA, 14–19 November 2021; SC '21. [CrossRef]
21. Oo, N.Z.; Chaikan, P. Power Efficient Strassen's Algorithm using AVX512 and OpenMP in a Multi-core Architecture. *ECTI Trans. Comput. Inf. Technol. (ECTI-CIT)* **2023**, *17*, 46–59. [CrossRef]
22. Jammal, F.; Aljabri, N.; Al-Hashimi, M.; Saleh, M.; Abulnaja, O. A Preliminary Empirical Study of the Power Efficiency of Matrix Multiplication. *Electronics* **2023**, *12*, 1599. [CrossRef]
23. Dasgupta, S.; Papadimitriou, C.; Vazirani, U. *Algorithms*, 1st ed.; McGraw-Hill Education: New York, NY, USA, 2006. Available online: http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf (accessed on 14 April 2024).
24. Aljabri, N.; Al-Hashimi, M.; Saleh, M.; Abulnaja, O. Investigating power efficiency of mergesort. *J. Supercomput.* **2019**, *75*, 6277–6302. [CrossRef]
25. Ghose, S.; Yaglikçi, A.G.; Gupta, R.; Lee, D.; Kudrolli, K.; Liu, W.X.; Hassan, H.; Chang, K.K.; Chatterjee, N.; Agrawal, A.; et al. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. *Proc. ACM Meas. Anal. Comput. Syst.* **2018**, *2*, 1–41. [CrossRef]
26. Free Software Foundation. A GNU Manual (3.11 Options That Control Optimization). 2021. Available online: https://gcc.gnu.org/onlinedocs/gcc-11.4.0/gcc/Optimize-Options.html (accessed on 26 April 2024).
27. AMD. AMD μProf. 2023. Available online: https://www.amd.com/en/developer/uprof.html (accessed on 4 August 2023).
28. AMD. AMD μProf User Guide, Rev 4. November 2022. Available online: https://www.amd.com/content/dam/amd/en/documents/developer/uprof-v4.0-gaGA-user-guide.pdf (accessed on 4 August 2023).
29. Lane, P.A.; Lobrano, J. The AMD Rome Memory Barrier. *arXiv* **2022**, arXiv:2211.11867.

30. Lu, J.; Zhang, J.; Tan, S.X.D. Real-time Thermal Map Estimation for AMD Multi-Core CPUs Using Transformer. In Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), San Francisco, CA, USA, 28 October–2 November 2023; pp. 1–7. [CrossRef]
31. Chakraborty, S.; Deb, D.; Buragohain, D.; Kapoor, H.K. Cache capacity and its effects on power consumption for tiled chip multi-processors. In Proceedings of the 2014 International Conference on Electronics and Communication Systems (ICECS), Coimbatore, India, 13–14 February 2014; IEEE: New York, NY, USA, 2014; pp. 1–6.