Exploratory Analysis of AVX-512 Kernel and Algorithmic Optimizations for DGEMM

Sahith Kancharla, sahith@vt.edu

***Abstract*—Dense matrix multiplication (DGEMM) is a key operation in many scientific and machine learning applications. This work explores the effects of algorithmic strategies and hardware-level optimizations on DGEMM performance on a 32-core Intel® Xeon® Gold 5218 processor. We compare a cache-blocked implementation of the standard algorithm, Strassen's method in both its original and hybrid forms, and analyze how each performs under OpenMP parallelization. Instead of focusing solely on timing results, we analyze the underlying causes of performance variation by collecting and analyzing hardware metrics such as cache behavior, memory traffic, and thread interactions. Additionally, we investigate the effects of loop unrolling and OpenMP scheduling on each approach. Using the insights gained from these experiments, we design a custom AVX-512 micro-kernel and integrate it into both the blocked and hybrid implementations. Detailed performance data is included in the appendix, supporting our analysis and offering practical guidance for optimizing dense matrix multiplication on modern multi-core systems.**

I. Introduction

Dense matrix multiplication (DGEMM) is a foundational operation in scientific computing, engineering simulations, and machine learning. It underpins a wide range of compute-intensive workloads such as numerical solvers, finite-element simulations, and deep learning inference pipelines. In particular, efficient double-precision matrix multiplication plays a key role in maximizing throughput across modern high-performance systems.

The growing complexity of contemporary hardware presents both new opportunities and challenges for performance tuning. Modern processors now feature wider SIMD units, such as AVX-512, deeper memory hierarchies, and increasing core counts. As a result, performance bottlenecks have shifted from raw floating-point throughput to bottlenecks associated with memory bandwidth, cache utilization, and parallel scalability. These changes demand a holistic view of performance, where data movement, reuse, and synchronization are often more important than arithmetic intensity alone.

Today's multi-core processors, including Intel's Xeon Gold platforms, are equipped with both high core counts and rich vector instruction sets. Leveraging these capabilities effectively requires not just theoretical algorithmic improvements but careful integration with low-level architectural features such as cache behavior, memory alignment, and SIMD vectorization. Libraries like OpenBLAS and Intel MKL encapsulate many of these optimizations, but they abstract away the underlying design decisions. To truly understand performance, one must examine how algorithmic structure interacts with hardware-specific details like instruction pipelines, NUMA layouts, and memory access patterns.

This paper presents an empirical study of multiple DGEMM strategies, designed to isolate the performance effects of algorithmic choices and low-level optimizations. We implement and evaluate three distinct approaches: a cache-blocked classical algorithm, which improves locality while retaining cubic arithmetic complexity; Strassen's recursive algorithm, which reduces multiplication count at the cost of memory overhead; and a hybrid variant that invokes blocked DGEMM below a problem-size cutoff. Each method is tested across serial and parallel configurations to analyze behavior at different levels of thread and memory contention.

In addition, we design a custom AVX-512 micro-kernel used across all implementations. This kernel is optimized for loop unrolling, register reuse, memory alignment, and instruction throughput. It enables a consistent performance foundation while allowing us to isolate the effects of recursion, blocking, and task granularity in higher-level algorithm variants.

This paper is organized around three guiding questions:

1) **Algorithm trade-offs:** How do the blocked, Strassen, and hybrid implementations differ in performance characteristics, cache usage, and scalability on a 32-core system?
2) **Runtime effects:** What impact do loop unrolling, recursion cutoff, and OpenMP task scheduling have on performance and cache efficiency?
3) **Kernel-level optimization:** How can low-level AVX-512 tuning improve throughput, and what interaction exists between micro-kernel design and the larger algorithm structure?

We vary matrix sizes, thread counts, recursion thresholds, and unrolling factors to quantify their effects using both wall-clock timing and hardware performance counters. In doing so, we provide a comprehensive analysis of how cache behavior, memory pressure, and parallel scaling shape the effectiveness of each algorithm. Our findings offer actionable insight into the principles behind efficient DGEMM on modern SIMD-enabled multicore systems, and extend to other compute-bound kernels where locality, synchronization, and memory traffic are key concerns.

II. Background and Related Work

A. Fundamental DGEMM Algorithms

The classical dense matrix–matrix multiplication algorithm uses a straightforward triple-nested loop structure, resulting in

$O(n^3)$ arithmetic complexity. Although pedagogically useful and easy to benchmark, this naïve method does not exploit modern processor caches or parallelism. As matrix sizes increase, performance quickly becomes bottlenecked by memory bandwidth and cache inefficiencies, making the approach inadequate for high-performance workloads on contemporary architectures [1]. These limitations have led to extensive research into improving both arithmetic intensity and memory behavior through tiling, loop unrolling, and data reuse strategies.

B. Strassen's Algorithm and Its Parallel Extensions

Strassen's algorithm reduces the number of scalar multiplications to $7, n^{\log_2 7} \approx n^{2.81}$ by substituting multiplications with additional additions [2]. While the asymptotic gain is theoretically compelling, practical performance is often limited by overheads in memory allocation and control flow. As a result, real-world benefits depend on hardware characteristics, recursion depth, and implementation strategy.

Recursion cutoff. Fully recursive Strassen implementations can become inefficient for small subproblems due to control overhead and memory fragmentation. Huss-Lederman *et al.* showed that introducing a recursion cutoff and switching to an optimized base-case kernel can significantly improve performance [3]. Further, Bilardi *et al.* found that stopping recursion near $n = 128$ yielded the best trade-off between arithmetic savings and memory efficiency in the ATLAS framework [4]. These findings support the design of hybrid strategies where recursion is truncated in favor of optimized leaf kernels.

Parallel mapping. In distributed-memory settings, Strassen's structure naturally produces seven independent subproblems per recursion level. The Communication-Avoiding Parallel Strassen (CAPS) algorithm by Lipshitz *et al.* achieved strong scaling when the number of processors matched powers of seven [5]. However, adapting this model to shared-memory environments presents challenges, including thread placement, memory allocation, and synchronization, which require specialized tuning and often diminish parallel efficiency at higher core counts.

C. Hybrid Strassen

Hybrid Strassen implementations combine recursion with traditional blocking in order to balance arithmetic complexity and practical execution performance. By cutting off recursion at a tunable threshold and using a cache-aware base kernel, these implementations aim to exploit Strassen's arithmetic savings while minimizing memory pressure [6]. Studies have shown that hybrid methods can provide $2\times$ speedups compared to full recursion, particularly when the base case leverages optimized SIMD kernels [7].

D. Cache-Aware Tiling and Blocking

Blocking divides matrices into smaller submatrices that fit into lower-level caches, reducing last-level cache (LLC) misses and enhancing spatial and temporal locality. This method is fundamental in high-performance libraries such as OpenBLAS and Intel MKL [1]. For shared-memory systems, Nimako *et al.* demonstrated that performance improves further when thread-to-block mappings are carefully coordinated and synchronization is minimized [8]. Blocking also facilitates loop unrolling and vectorization, which are essential for high throughput on modern CPUs.

E. Shared-Memory Parallel Frameworks

OpenMP remains widely adopted for shared-memory parallelism, but basic loop-level parallelism often yields limited benefits due to load imbalance and synchronization overhead. Bentz and Kendall observed that a naïve OpenMP implementation of DGEMM delivered modest speedup in scientific workloads [9], while Hackenberg *et al.* showed that well-tuned OpenMP code could achieve gains close to those of commercial libraries [10]. Task-based runtimes such as QUARK and PLASMA provide more dynamic scheduling and dependency tracking, which improve load balancing for irregular workloads. SRUMMA builds on these ideas in hybrid architectures, although its task management introduces extra overhead [11], [12].

F. SIMD Vectorization and AVX Extensions

Contemporary CPUs support wide SIMD units such as SSE, AVX, and AVX-512, enabling multiple floating-point operations per cycle. Fully exploiting these capabilities requires low-level optimizations including register tiling, loop restructuring, and memory alignment. While compilers provide auto-vectorization support, recent work has shown that manually optimized AVX-512 kernels often outperform compiler-generated code in bandwidth-constrained scenarios [13], [7]. Tools such as Intel VTune and Linux `perf` counters are valuable in identifying performance hotspots and guiding vectorization.

Synopsis. Multiple approaches exist for optimizing DGEMM, including classical triple-loop formulations, cache-aware blocking, Strassen-style recursion, shared-memory scheduling, and SIMD-level tuning. Each approach presents different trade-offs depending on workload size, thread count, and system architecture. This study synthesizes these methods into a unified implementation pipeline designed for 32-core AVX-512 systems. By combining blocking, hybrid recursion, and AVX-512 micro-kernels, we aim to understand and improve the performance of dense matrix multiplication across algorithmic and architectural layers.

III. Methodology

A. Software Environment

All DGEMM kernels in this study are implemented in C and parallelized using OpenMP 4.5. The innermost loops leverage AVX-512 intrinsics for vectorized computation. To optimize for target hardware, we compile with `gcc 11.5` using the flags `-O3 -ffast-math -fopenmp -march=skylake-avx512`, which enable aggressive optimization and explicit targeting of the Skylake AVX-512 microarchitecture. These settings are intended to maximize performance by ensuring that the compiler generates

highly tuned code for vectorization, loop transformation, and aligned memory access on the target hardware.

B. Algorithms Under Test

To explore the trade-offs between algorithmic complexity, cache behavior, and parallel scalability, we evaluate four distinct DGEMM implementations:

- **Naïve DGEMM:** A simple implementation using three nested loops with cubic arithmetic complexity ($O(n^3)$), offering a clear performance baseline but lacking any form of blocking or SIMD acceleration.
- **Blocked DGEMM:** A cache-aware approach that partitions the computation into $b \times b$ blocks to enhance locality within the L1 cache. We experiment with block sizes $b = 32$ to understand their effects on cache reuse and performance.
- **Strassen's Algorithm:** A recursive algorithm that reduces the arithmetic complexity to $O(n^{\log_2 7})$ by decreasing the number of multiplications at the cost of additional additions and intermediate storage. We implement it recursively down to 1×1 base cases.
- **Hybrid Strassen:** A variant that limits the recursion depth by introducing a cutoff size n_{cut}. Subproblems at or below this size are delegated to the blocked DGEMM kernel. We evaluate $n_{\text{cut}} = 128$ to determine its influence on both computation and memory traffic.

Although faster matrix multiplication algorithms like Winograd's method exist, their numerical instability and high implementation complexity often make them unsuitable for general-purpose use. Prior studies show they suffer from coefficient growth and rounding errors[14]. Thus, we focus on Strassen-like recursion, which offers a more practical trade-off.

C. Hardware Platform and Performance Monitoring

All experiments are performed on a dual-socket Intel Xeon Gold 5218 system (Cascade Lake, 2.3 GHz) with 32 physical cores in total. This system supports AVX-512 instructions and provides a balanced memory bandwidth profile suitable for cache-sensitive workloads. Timing measurements are captured using `omp_get_wtime` to provide precise wall-clock execution durations. To assess cache efficiency and memory behavior, we use Linux `perf` to gather hardware performance counters, including LLC load and store misses, total cache references, and other relevant metrics. Each experiment is repeated 14 times, with the two highest and two lowest measurements discarded to mitigate outliers. The average of the remaining 10 runs is reported to ensure statistical stability.

D. Key Metrics

We analyze each algorithm variant using the following performance and system-level metrics:

- **Elapsed Time:** The total time taken for matrix multiplication as measured on the wall clock.
- **Speedup:** The relative speedup compared to the single-threaded naïve and naïve-unrolled baselines, which helps normalize performance gains across implementations.
- **Cache Behavior:** Metrics such as LLC load misses and total cache accesses, collected via `perf`, to quantify how effectively each algorithm utilizes the cache hierarchy.
- **Memory Footprint:** An estimate of the memory working set size, inferred from cache miss counts and memory traffic. This metric is especially relevant for recursive algorithms that allocate temporary buffers.

E. Optimization Techniques

To isolate the impact of low-level performance strategies, we apply the following optimizations uniformly across relevant variants:

- **Loop Unrolling:** The innermost loop over k is unrolled by a factor of four to reduce control overhead, increase instruction-level parallelism, and improve throughput.
- **Cache Blocking:** We tune the block sizes to fit within L1 cache depending on the configuration, striking a balance between cache utilization, TLB behavior, and loop nesting overhead.
- **Software Prefetching:** Using `__mm_prefetch` intrinsics, we preemptively fetch memory into cache to overlap memory latency with computation and improve instruction throughput.
- **Register Blocking:** Matrix panels of A are packed into contiguous memory regions to support aligned loads, while elements of B are broadcast into ZMM registers to exploit the full width of AVX-512 instructions.

F. AVX-512 Micro-Kernel Implementation

The AVX-512 micro-kernel serves as the performance-critical core for both the blocked and hybrid Strassen implementations. The primary kernel operates on a 24×6 tile of the output matrix C, and its structure is as follows:

- Each column of C is backed by three ZMM registers, which hold 8, 8, and 8 elements respectively, allowing simultaneous updates to 24 rows across 6 columns.
- Elements from B are broadcast into ZMM registers using scalar-to-vector expansion, enabling efficient reuse across multiple rows of A.
- Each iteration of the k loop performs six fused multiply-add (FMA) operations across the loaded vectors using the `_mm512_fmadd_pd` intrinsic.
- The loop is manually unrolled by a factor of two to minimize loop overhead and maximize ILP (instruction-level parallelism). Further unrolling is feasible with stride adjustments and FMA pipeline tuning.
- For boundary conditions (mr < 24), AVX-512 mask-enabled load/store instructions ensure safe memory accesses while maintaining vector lane utilization.

For the Strassen AVX implementation, a separate 16×8 kernel is employed. This choice is guided by the recursive structure of the algorithm and the experimental test range, where the base-case matrices are consistently 128×128. The 16×8 tile ensures full tile coverage of the base case without edge handling, improving both register usage and cache

TABLE I
FAST-PATH STEPS FOR THE 24×6 AVX-512 MICRO-KERNEL (MR = 24, NR = 6)

Step	Code Pattern	Description
1	`if (mr == 24 && nr == 6) { ... }`	Fast-path guard for exact tile size. Avoids masks and selects optimized code path.
2	`__m512d C0[3], ..., C5[3];`	Declare 6 accumulators for columns 0–5, each storing 3 contiguous 8-element rows (total 24 rows).
3	`C0[0] = _mm512_load_pd(C + 0);` ... `C5[2] = _mm512_load_pd(C + 5*ldC + 16);`	Load each column's values from memory into 3 aligned 512-bit registers (24 values per column).
4	`for (int p = 0; p + 1 < kc; p += 2) { ... }`	Main loop over the shared dimension (kc), unrolled in steps of 2 with prefetching and fallback for odd iteration.
4a	`a0 = _mm512_load_pd(A + 0);` `a1 = _mm512_load_pd(A + 8);` `a2 = _mm512_load_pd(A + 16);`	Load 3 slices (8 elements each) from A matrix for the current kc index.
4b	`b0 = _mm512_set1_pd(B[0]);` ... `b5 = _mm512_set1_pd(B[5]);`	Broadcast scalar elements from B into full 512-bit registers for FMA.
4c	`C0[0] = _mm512_fmadd_pd(a0, b0, C0[0]);` ... `C5[2] = _mm512_fmadd_pd(a2, b5, C5[2]);`	Fused multiply-add over 6 columns × 3 slices. Each broadcasted B value is multiplied with A and added to C.
4d	`A += 24; B += 6;`	Advance pointers to A and B for the next kc iteration.
5	`_mm512_store_pd(C + 0, C0[0]);` ... `_mm512_store_pd(C + 5*ldC + 16, C5[2]);`	Store updated accumulators from registers back into matrix C (aligned store).
6	`return;`	Exit the fast path. Remaining cases (e.g., mr ¡ 24) handled separately using masks.

alignment. This configuration was empirically determined to provide superior performance for the recursive base size while preserving high SIMD occupancy and data locality.

This micro-kernel design balances register pressure, instruction throughput, and memory bandwidth, and is adaptable to both regular blocked and recursive Strassen-style matrix multiplication.

The source code, configuration scripts, and data collection tools used in this study are publicly available at the following repository: GitHub Repository.

IV. RESULTS AND ANALYSIS

A. Strassen's Algorithm

Strassen's method reduces the arithmetic complexity of DGEMM, offering theoretical improvements by minimizing the number of multiplications. However, its implementation introduces substantial overhead due to the recursive structure and the reliance on auxiliary temporary matrices.

When executed without a recursion cutoff, the parallel version of the algorithm performs consistently worse than its serial counterpart across all tested matrix sizes (Table II). For example, the runtime for $n = 1024$ increases dramatically from 3.69 seconds with two threads to 22.99 seconds with 32 threads. This significant performance degradation stems from several architectural and algorithmic challenges:

- **Excessive recursion:** The algorithm generates more than *1.6 million* recursive calls (Table III), with each call averaging just 0.002 microseconds. Although each call is brief, their cumulative cost burdens performance and may overwhelm the call stack.
- **Allocator pressure:** The number of memory allocations exceeds *20 million* for $n = 2048$, consuming nearly 4 tebibytes (TiB) of temporary memory (Table IV). This leads to frequent cache invalidations, translation lookaside buffer (TLB) misses, and poor memory locality.
- **Increased stack usage:** Stack memory usage rises from 304 bytes at $n = 64$ to 832 bytes at $n = 512$ (Table III). This increase exacerbates NUMA traffic and raises the risk of page faults in multi-socket systems.
- **Task scheduler overload:** Without a base-case cutoff, the algorithm spawns an excessive number of fine-grained OpenMP tasks. Beyond seven threads, no additional parallelism is available at the top recursion level. Increasing the thread count beyond this point leads to greater contention and inefficient thread utilization.

To address these limitations, a base-case cutoff is introduced at $n = 128$. Below this threshold, the algorithm switches to a cache-blocked DGEMM kernel. This hybrid approach reduces the number of recursive calls at $n = 128$ from over 47 million to just 1654 for $1024x1024$ matrix (Table V), and cuts memory allocations by more than 97% (Table VI). In single-threaded scenarios, this variant yields up to a $2\times$ speedup for medium-sized matrices in the range $128 < n \leq 512$. However, for $n > 1024$, performance gains diminish due to increased last-level cache (LLC) misses and store contention, especially when loop unrolling is applied. Even with the base-case cutoff, Strassen's algorithm exhibits poor scalability on 32-core systems and is therefore excluded from our final performance recommendations.

a) Speedup.: The parallel speedup of Strassen's algorithm is non-monotonic, as shown in Tables XVI through XXIV. A notable improvement is observed when increasing the thread count from six to seven. This pattern arises from the algorithm's structure, which recursively splits each

problem into seven independent subproblems. This behavior aligns with the analysis by Lipshitz and Ballard [5]. However, no further speedup is achieved beyond seven threads, even though additional cores are available. This plateau results from the fact that only seven tasks can be executed concurrently at the top level of recursion. Deeper recursive levels do not provide sufficient parallelism to offset the associated overhead. Consequently, Strassen performs well at lower thread counts but becomes increasingly inefficient relative to blocked DGEMM as the number of cores increases.

b) Cache Behavior.: Each level of recursion in Strassen's algorithm introduces new temporary matrices of size $n/2 \times n/2$, significantly increasing the working set. For instance, during serial execution at $n = 512$, the hybrid Strassen variant incurs 0.27 million LLC load misses, compared to just 0.04 million for the blocked DGEMM implementation (Table IX). As matrix size increases, the number of LLC store misses in Strassen significantly exceeds that of blocked DGEMM, primarily due to frequent writes to temporary buffers. Under parallel execution, cache pressure intensifies further, compounding the performance penalty.

c) Memory Footprint.: Strassen's recursive formulation demands substantial auxiliary memory. Profiling reveals that the hybrid Strassen variant requires approximately 0.6 GiB of additional workspace, which exceeds the combined size of the input and output matrices. In contrast, blocked DGEMM maintains a compact and bounded memory footprint, typically needing only the result matrix and a small number of cache-aligned panels of size $n \times b$. The large memory requirement in Strassen leads to frequent cache evictions and elevated NUMA traffic, ultimately reducing its efficiency on multicore platforms.

d) Additional Observations.: Empirical results confirm that a pure implementation of Strassen's algorithm, executed without a base-case cutoff, is not suitable for practical parallel computation. As matrix dimensions increase, the algorithm generates billions of extremely fine-grained tasks. This behavior overwhelms cache resources, saturates the task scheduler, and, in some instances, causes runtime failures due to exceeding system recursion or allocation limits. For these reasons, we do not pursue the pure recursive Strassen variant further in this study.

B. Blocked DGEMM: Cache Tiling Effects

Cache-aware tiling partitions the $n \times n$ matrix multiplication into smaller $b \times b$ submatrices sized to fit within the L1 caches, significantly improving data locality. In a serial execution for matrix size $n = 1024$, applying cache-tiling reduces last-level cache (LLC) load misses from 4M in the naïve implementation to 1.6M, representing a reduction of approximately 59% (Table IX). Although smaller matrices ($n < 256$) exhibit additional overhead due to block indexing computations, blocking consistently achieves the lowest LLC miss rates for larger matrices ($n \geq 256$), laying a robust foundation for efficient parallel execution.

a) Speedup.: In single-threaded experiments (Table X), the blocked DGEMM algorithm achieves a 2.5× speedup over the naïve algorithm at $n = 1024$, reducing execution time from 3.179s to 1.229s. Under an eight-thread execution scenario (Table XX), the blocked kernel achieves a speedup of 6× relative to its serial counterpart, while the blocked and unrolled variant reaches an ideal 7.9x improvement over its serial implementation.

b) Cache Behavior.: Blocking not only reduces LLC load misses, but it also significantly lowers store-related traffic. For serial runs at matrix size $n = 1024$, LLC store misses decrease from 4,027,132 in the naïve implementation to 1,679,236 with blocking (Table IX). Under eight-thread conditions (Table XIX), blocked and unrolled execution experiences only a modest 30% increase in LLC load misses compared to serial runs. This resilience under higher thread counts underscores blocking's effectiveness in mitigating cache thrashing in multithreaded environments.

c) Memory Footprint.: The additional memory overhead introduced by blocking is minimal, consisting primarily of two temporary $b \times b$ panels per thread, one for each input matrix, resulting in a memory requirement of $O(b \cdot n)$ additional space. For example, this amounts to approximately 1 MiB per thread with a block size $b = 64$. This modest footprint contrasts sharply with the significantly larger recursive temporary buffers required by Strassen's algorithm. Consequently, blocked DGEMM experiences lower cache eviction rates and reduced NUMA-related memory traffic, enhancing performance in shared-memory architectures.

d) Synchronization Overhead.: Since individual blocks are computed independently, thread synchronization occurs infrequently and primarily at the upper loop levels managed by OpenMP. This minimal synchronization cost, coupled with excellent cache reuse, renders blocked DGEMM particularly efficient in both single-threaded and highly parallelized contexts.

C. Loop Unrolling

Unrolling the innermost k-loop by a factor of four significantly reduces branch and loop-control overhead while exposing additional instruction-level parallelism. Table X and Table IX quantify these benefits across implementations:

- **Naïve DGEMM:** For $n = 1024$, unrolling decreases execution time from 3.178 s to 0.425 s, representing a 6× improvement, and lowers LLC load misses by a significant margin (from 38639M to 2009M).
- **Blocked DGEMM:** while loop unrolling is less effective on large matrices in serial execution; however, as the number of threads increases, it significantly contributes to lower wall times and also significantly last last-level cache misses.
- **Hybrid Strassen:** With a cutoff at $n = 128$, unrolling cuts the runtime in half at $n = 512$ (from 0.682828 s to 0.308256 s, a 1.93× speedup), while decreasing LLC load misses by 35%

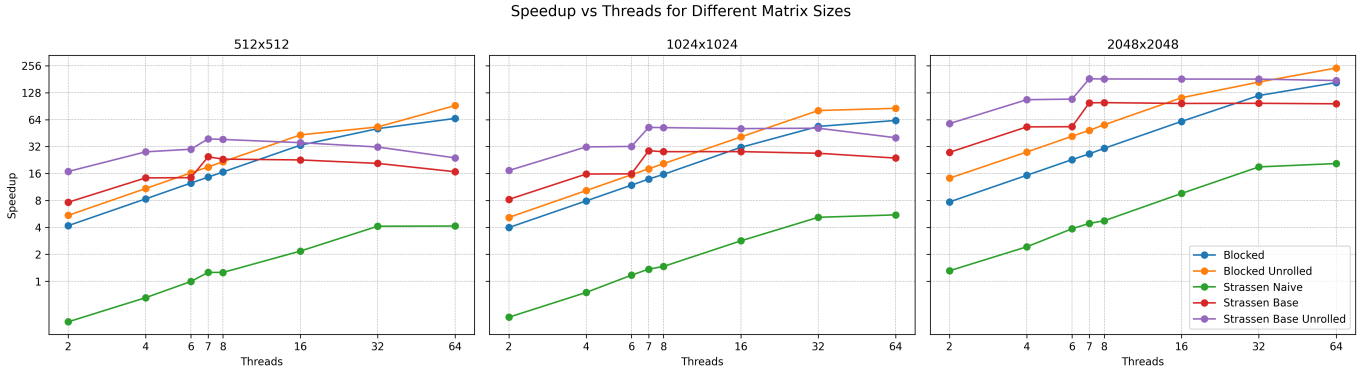

Fig. 1. Parallel speedup vs. thread count for matrix sizes 512, 1024, and 2048 against serial naive execution time. Note how Blocked and Blocked Unrolled implementations show nearly linear scaling, while Strassen variants saturate earlier.

D. Performance Comparison Summary

Our comprehensive evaluation identifies distinct performance regimes:

- **Small matrices** ($n \leq 128$)**:** The overhead introduced by blocking and unrolling outweighs any performance benefit. In this regime, the naïve or unrolled naïve implementations are most efficient.
- **Medium matrices** ($128 < n \leq 512$)**:** In serial execution, hybrid Strassen with unrolling outperforms other methods by up to 1.5× compared to blocked plus unrolled, capitalizing on its reduced multiplication complexity while containing memory overhead.
- **Large matrices** ($n > 512$)**:** Blocked plus unrolled DGEMM delivers the highest absolute throughput and the lowest LLC miss rates, due to superior cache reuse and minimal auxiliary storage requirements.
- **Parallel scaling (up to 32 cores):** Blocked plus unrolled DGEMM sustains the best speedup across all thread counts, achieving nearly linear scaling (e.g., 30× speedup on 32 threads for $n = 2048$). In contrast, hybrid Strassen performs well at lower thread counts (up to 7), but its performance plateaus beyond this point due to limited parallelism and greater cache contention.

In summary, on our 32-core AVX-512 platform, the combination of cache tiling and loop unrolling consistently delivers the optimal balance of raw performance, cache efficiency, and parallel scalability across a broad spectrum of matrix sizes.

V. AVX Implementation Analysis

A. Blocked AVX-512 Implementation

To fully leverage SIMD capabilities, the blocked DGEMM kernel is implemented using AVX-512 intrinsics with a carefully tuned micro-kernel that operates on 24×6 output tiles. This layout ensures high register utilization and minimizes instruction overhead through FMA operations and loop unrolling. All intermediate buffers are aligned to 64-byte boundaries to match cache line and register widths, facilitating fast aligned loads and stores.

This section focuses specifically on AVX integration, assuming familiarity with the blocking principles discussed earlier. SIMD execution, combined with loop unrolling and prepacked memory layouts, sustains high throughput even under multithreaded conditions. For large matrices ($n \geq 2048$), the kernel reaches a peak speedup of over 2450× relative to the original naïve baseline and 200× over the unrolled scalar variant. These gains result from AVX-512's high computational density and the reduced synchronization overhead due to cache-aware tiling.

B. Hybrid Strassen with AVX-512 Kernel

This variant uses Strassen's recursive decomposition, falling back to the AVX-512 kernel at the base case. A cutoff at $n = 128$ ensures the base case remains large enough to exploit vectorization, while higher levels apply recursive partitioning. Each recursion level reuses pre-allocated buffers to reduce allocation overhead.

Although the hybrid design benefits from fewer multiplications, the costs of managing temporary data and increased cache pressure offset these savings at scale. Parallel performance is also constrained by Strassen's limited top-level concurrency. As in earlier results, performance plateaus beyond 7 threads, with the variant achieving a 45× speedup on 32 cores relative to the unrolled baseline—substantial, but notably lower than the pure blocked approach.

Moreover, the transition point between Strassen recursion and AVX-512 DGEMM introduces non-trivial overhead. Even though the recursive structure reduces multiplication count, the context switching between Strassen's arithmetic and the micro-kernel incurs synchronization, packing, and memory management costs. These transition costs become significant at scale and erode the arithmetic advantage. This trend is quantitatively reflected in Appendix Table XXVII (e.g., Tables IX and XIX), where cache misses and store traffic for hybrid Strassen remain consistently higher than those of the pure blocked approach. Thus, while hybridization enhances serial performance in mid-size regimes, its scalability is ultimately bottlenecked by memory traffic, recursion overhead, and the base-case kernel mismatch.

VI. Conclusion

We have presented a comprehensive study of dense DGEMM on a 32-core AVX-512 platform, comparing pure blocked, loop-unrolled, and hybrid Strassen implementations. Our custom 24×6 micro-kernel with a 2× unroll of the inner k-loop-delivers exceptional performance: it sustains near-peak FMA throughput, minimizes LLC misses, and scales almost linearly to 32 threads. Loop unrolling further reduces branch and loop-control overhead, while cache-aware tiling ensures high data locality and low synchronization costs.

The hybrid Strassen variant, which applies Strassen's $O(n^{\log_2 7})$ recursion down to a cutoff of $n = 128$, shows modest serial gains but fails to compete at large sizes or high thread counts with pure blocked implementation. Its overhead of auxiliary buffers, recursive management, and limited top-level task parallelism leads to increased LLC misses and stalls Strassen at a 49× speedup on 32 cores, far below our blocked micro-kernel.

For small matrices ($n \leq 128$) to truly large problems ($n \geq 8192$), the combination of cache tiling and a tuned 24×6 micro-kernel with loop unrolling consistently offers the best balance of raw throughput, cache efficiency, and parallel scalability. Consequently, we recommend the pure blocked + unrolled DGEMM as the default dense multiplication method on modern multi-core AVX-512 CPUs for the 32-core node over rlogin.

A. Future Work

Future research could extend these optimization strategies in several directions. First, adapting the presented methods to support mixed-precision arithmetic may yield significant benefits, particularly for machine learning and scientific workloads where precision requirements vary. Additionally, exploring deeper register-level blocking and instruction scheduling may help uncover further performance gains from AVX-512 and future SIMD extensions. On the architectural side, integrating NUMA-aware scheduling and memory placement policies can improve scalability and memory efficiency on multi-socket systems. Another promising avenue is the development of adaptive, runtime-tunable kernels that dynamically select the most efficient algorithm based on matrix dimensions, thread count, and hardware topology. Finally, building a lightweight autotuning framework or runtime advisor that profiles the target platform and recommends an optimal DGEMM strategy could help generalize these findings into a portable, high-performance solution.

References

[1] J. J. Dongarra, J. D. Cruz, S. Hammarling, and I. S. Duff, "Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 18–28, 1990.

[2] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[3] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull, "Implementation of strassen's algorithm for matrix multiplication," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC '96)*. IEEE, 1996, p. 32. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/369028.369096

[4] G. Bilardi, Luigi D'Alberto, and A. Nicolau, "Using recursion to boost atlas's performance," University of California, Irvine, Tech. Rep. ICS TR 04-01, 2004. [Online]. Available: https://ics.uci.edu/~paolo/Reference/paoloA.ishp-vi.pdf

[5] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE, 2012, pp. 1–13. [Online]. Available: https://www.cs.huji.ac.il/~odedsc/papers/CAPS-impl.pdf

[6] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson, "Implementation of strassen's algorithm for matrix multiplication," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 32–es.

[7] M. Coppola, M. Danelutto, and M. Vanneschi, "Performance prediction and evaluation of parallel algorithms for matrix multiplication," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1445–1465, 2008.

[8] G. Nimako, E. J. Otoo, and D. Ohene-Kwofie, "Fast parallel algorithms for blocked dense matrix multiplication on shared memory architectures," in *Algorithms and Architectures for Parallel Processing: 12th International Conference, ICA3PP 2012, Fukuoka, Japan, September 4-7, 2012, Proceedings, Part I 12*. Springer, 2012, pp. 443–457.

[9] J. L. Bentz and R. A. Kendall, "Parallelization of General Matrix Multiply Routines Using OpenMP," in *Proceedings of the International Workshop on OpenMP Applications and Tools (WOMPAT)*, ser. Lecture Notes in Computer Science, B. M. Chapman, Ed., vol. 3349. Springer, 2005, pp. 1–11.

[10] D. Hackenberg, R. Schöne, W. E. Nagel, and S. Pflüger, "Optimizing OpenMP Parallelized DGEMM Calls on SGI Altix 3700," in *Euro-Par 2006 Parallel Processing*, ser. Lecture Notes in Computer Science, W. E. Nagel, W. V. Walter, and W. Lehner, Eds., vol. 4128. Springer, 2006, pp. 145–154.

[11] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling Dense Linear Algebra Operations on Multicore Processors," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 1, pp. 15–44, 2010.

[12] M. Krishnan and J. Nieplocha, "SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2004, pp. 987–996.

[13] Y. Liu, X. Wang, and H. Zhang, "Benchmarking avx-512 optimizations for dense matrix multiplication," *Journal of High Performance Computing*, vol. 34, no. 2, pp. 123–135, 2020.

[14] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-avoiding algorithms for linear algebra and beyond," in *Proceedings of the IEEE*, vol. 101, no. 11. IEEE, 2013, pp. 1874–1897.

APPENDIX

TABLE II
PARALLEL STRASSEN NAÏVE AVERAGE TIMES (SECONDS)

# Threads	64×64	128×128	256×256	512×512
2	0.010856	0.071440	0.516047	3.691921
4	0.015650	0.133573	1.596175	11.141655
6	0.036234	0.248065	1.847483	13.149373
7	0.039044	0.266003	2.035092	13.739610
8	0.040524	0.290303	1.979509	13.784145
16	0.053967	0.351649	2.390789	17.265195
32	0.077128	0.490285	3.411832	22.997039

TABLE III
STRASSEN RECURSIVE CALL COUNTS

n	Calls	Time (s)	Avg time/call (μs)	Max stack (bytes)
64	19,608	0.000036	0.002	304
128	137,257	0.000245	0.002	352
256	960,800	0.001728	0.002	400
512	6,725,601	0.011948	0.002	448
1024	47,079,208	0.083620	0.002	496
2048	329,554,457	0.585064	0.002	544
4096	2,306,881,200	4.082900	0.002	592
8192	16,148,168,401	28.531665	0.002	640

TABLE IV
STRASSEN ALLOCATION & MEMORY COUNTS

n	Init calls	Total bytes	Size (MiB)	Max stack (bytes)
64	81,230	4,914,976	4.69	348
128	568,633	35,256,800	33.62	448
256	3,980,454	250,205,472	238.61	512
512	27,863,201	1,765,069,792	1,683.30	576
1024	195,042,430	12,410,014,496	11,835.11	640
2048	1,365,297,033	87,088,205,280	83,053.78	704
4096	9,557,079,254	610,489,852,192	582,208.49	768
8192	66,899,554,801	4,276,918,626,272	4,078,787.45	832

TABLE V
STRASSEN RECURSIVE CALL COUNTS WITH 128 AS BASE CASE

n	Calls	Time (s)	Avg Time/Call (µs)	Max Stack (bytes)
64	1	0.000000	0.189	1
128	1	0.000000	0.039	1
256	8	0.000000	0.027	106
512	57	0.000001	0.009	106
1024	400	0.000002	0.004	266
2048	2801	0.000009	0.003	266
4096	19608	0.000056	0.003	426
8192	137257	0.000378	0.003	426

TABLE VI
STRASSEN ALLOCATION & MEMORY COUNTS WITH 128 AS BASE CASE

n	Init Calls	Total Bytes	Size (MiB)	Max Stack (bytes)
64	1	32,768	0.03	1
128	1	131,072	0.12	1
256	30	4,325,376	4.12	69
512	233	43,909,120	41.88	133
1024	1654	361,889,792	345.12	197
2048	11601	2,751,332,352	2623.88	261
4096	81230	20,131,741,696	19199.12	325
8192	568633	144,411,852,800	137721.88	389

TABLE VII
STRASSEN RECURSIVE CALL COUNTS (32 THREADS) WITH 128 AS BASE CASE

n	Calls	Time (s)	Avg Time/Call (µs)	Max Stack (bytes)
64	1	0.000000	0.204	1
128	1	0.000000	0.047	1
256	8	0.000002	0.217	511
512	57	0.000001	0.024	975
1024	400	0.000007	0.016	1439
2048	2801	0.000043	0.015	1903
4096	19608	0.000292	0.015	2367
8192	137257	0.002032	0.015	2831

TABLE VIII
STRASSEN ALLOCATION & MEMORY COUNTS (32 THREADS) WITH 128 AS BASE CASE

n	Init Calls	Total Bytes	Total (MiB)	Max Stack (bytes)
64	1	32,768	0.03	1
128	1	131,072	0.12	1
256	30	4,325,376	4.12	527
512	233	43,909,120	41.88	1023
1024	1654	361,889,792	345.12	1519
2048	11601	2,751,332,352	2623.88	2015
4096	81230	20,131,741,696	19199.12	2511
8192	568633	144,411,852,800	137721.88	3007

TABLE IX
CACHE AND MEMORY METRICS FOR SERIAL IMPLEMENTATION

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
naive	64	16505	246383	49523	2746	32150	<notcounted>
naive_unrolled	64	11329	319170	50595	1994	14165	2332
blocked	64	9385	238857	50701	3332	12732	4765
blocked_unrolled	64	7392	267414	49054	1509	12658	234
strassen	64	10998	313787	80034	2704	4695	966
strassen_base	64	12760	253477	52535	2853	22382	<notcounted>
strassen_base_unrolled	64	11253	262333	53112	2680	<notcounted>	<notcounted>
naive	128	32716	352655	79497	10043	3688	1200
naive_unrolled	128	12574	335798	74976	2993	3023	1301
blocked	128	12900	244759	71816	4114	10493	1806
blocked_unrolled	128	21396	385885	78304	4169	4899	2066
strassen	128	31558	707732	130609	7362	65728	4036
strassen_base	128	17762	416392	95850	9461	20966	2245
strassen_base_unrolled	128	13196	342328	85023	8739	30770	3279
naive	256	55032	816038	264099	18770	14175	5716
naive_unrolled	256	88124	1035649	192395	19037	20872	5508
blocked	256	51768	1025770	253090	20726	11391	3797
blocked_unrolled	256	53120	963165	245956	20839	16938	3211
strassen	256	125964	3800094	721271	23880	454003	32155
strassen_base	256	99746	2379219	468491	22365	270098	12437
strassen_base_unrolled	256	82255	2387549	494166	20602	241330	12795
naive	512	470791	1152204226	1148344120	529292	228041	23901
naive_unrolled	512	216458	167996866	23844657	62091	162744	36952
blocked	512	836633	37165167	4329617	167064	78286	42711
blocked_unrolled	512	488726	37811497	4660633	119737	61200	41500
strassen	512	2678788	26553404	5660827	794470	3930703	268255
strassen_base	512	1876355	17003004	3799668	558578	1983254	244358
strassen_base_unrolled	512	1892323	17642366	3826400	558635	2256410	307357
naive	1024	6582859	10762543974	10753808804	4027132	1366056	263647
naive_unrolled	1024	4178101	1358966599	169109715	1619394	978686	212231
blocked	1024	8397172	298828245	30772804	1679236	380372	175946
blocked_unrolled	1024	8068292	295827952	32400873	1938849	487486	198367
strassen	1024	28143035	196516368	40002254	8686734	31045746	3732836
strassen_base	1024	21425221	133395821	28812161	6986163	19363320	2961229
strassen_base_unrolled	1024	20333954	139020922	34969969	7654840	12743873	2061898
naive	2048	38554799397	82331997713	82078645219	38639600914	8888636	4813644
naive_unrolled	2048	4060723677	10932346207	2855980410	2009059291	23581826	1392752
blocked	2048	1018433694	2563490306	283161797	114508839	2778497	1420890
blocked_unrolled	2048	1074303573	2576413818	297067557	118482291	2959000	1469790
strassen	2048	256318752	1385256497	293967319	75364421	223016360	29549146
strassen_base	2048	223421477	970410403	205551458	61466365	135647866	26188708
strassen_base_unrolled	2048	210414248	969153657	211041491	61146226	141928467	24312938

TABLE X
AVERAGE EXECUTION TIME FOR SERIAL IMPLEMENTATIONS (SECONDS)

Size	**Naive**	**Naive_Unrolled**	**Blocked**	**Blocked_Unrolled**	**Strassen**	**Strassen_Base**	**Strassen_Base_Unrolled**
64	0.000142	0.000077	0.000232	0.000176	0.003212	0.000152	0.000086
128	0.001687	0.000598	0.001886	0.001425	0.022549	0.001808	0.000785
256	0.014919	0.005352	0.015898	0.011828	0.166979	0.014160	0.006677
512	0.417953	0.055319	0.154443	0.154028	1.093155	0.098076	0.044940
1024	3.177837	0.425128	1.229078	1.233816	7.641909	0.682828	0.308256
2048	75.405665	5.869505	10.440503	10.643851	53.536960	4.801258	2.152973

TABLE XI
CACHE AND MEMORY METRICS (2 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	15147	302441	80331	6626	5697	409
blocked_unrolled	64	13225	279028	84203	5516	3279	83
strassen	64	21525	441713	92964	5209	9953	725
strassen_base	64	15095	280967	82132	5310	2570	546
strassen_base_unrolled	64	7042	222304	69518	8714	6397	371
blocked	128	11634	352812	99725	9408	15767	2423
blocked_unrolled	128	12710	308637	85442	8683	31141	2392
strassen	128	50026	868186	146281	12532	79532	5915
strassen_base	128	19323	402035	102971	10702	34696	4309
strassen_base_unrolled	128	23314	554718	117887	9058	30485	2689
blocked	256	71875	1483537	299884	15100	34968	8573
blocked_unrolled	256	39711	1347480	278949	17022	28436	4543
strassen	256	134953	4824455	832345	56036	709460	20224
strassen_base	256	97932	3200271	636011	26990	359223	16019
strassen_base_unrolled	256	53551	3146737	712757	24506	307885	8778
blocked	512	466021	50886224	5865694	90012	175002	34707
blocked_unrolled	512	587276	52728706	6290078	118901	154172	30826
strassen	512	4278546	41754204	7239126	1087134	5449923	782700
strassen_base	512	2604380	23976911	5090659	804104	3156422	339901
strassen_base_unrolled	512	2590113	24819174	5176918	805351	3277221	405611
blocked	1024	9122018	431536639	46742771	1913741	770474	268714
blocked_unrolled	1024	9168870	423393056	46893334	2630265	827871	259352
strassen	1024	47637520	283615606	54907988	16150495	38780714	4156083
strassen_base	1024	37301430	184458191	36463256	11592441	25271217	4443228
strassen_base_unrolled	1024	39231608	194822130	41292939	11555028	22723757	4029824
blocked	2048	1323366145	3704758902	371775095	120825328	12826334	2511221
blocked_unrolled	2048	1252204204	3632524142	418883429	115122374	4238159	2203884
strassen	2048	390545465	1996090535	385922476	126589427	288991476	33274379
strassen_base	2048	348163891	1360432265	284736335	107297821	175699886	32321042
strassen_base_unrolled	2048	341725954	1353021455	291132897	109202037	174910212	31843985

TABLE XII
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (2 THREADS)

Size	Blocked	Blocked_Unrolled	Strassen	Strassen_Base	Strassen_Base_Unrolled
64	0.000197	0.000091	0.006629	0.000155	0.000099
128	0.001557	0.000722	0.046334	0.001799	0.000784
256	0.012426	0.007175	0.184681	0.008260	0.003852
512	0.099851	0.076603	1.186449	0.054681	0.024794
1024	0.798076	0.615000	8.020625	0.386530	0.183716
2048	9.791930	5.313415	57.599675	2.734615	1.304559

TABLE XIII
CACHE AND MEMORY METRICS FOR PARALLEL IMPLEMENTATION (4 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	67581	308450	80222	25982	30479	5047
blocked_unrolled	64	53975	285432	89768	30703	20174	2495
strassen	64	21339	386732	101736	11701	12379	2061
strassen_base	64	25874	360263	83286	7156	1511	385
strassen_base_unrolled	64	7645	253358	67845	8454	9721	644
blocked	128	146203	644624	149738	32913	9771	4436
blocked_unrolled	128	119875	601031	153630	34898	10314	4350
strassen	128	88764	972122	179802	25675	74688	9641
strassen_base	128	31234	629938	134298	8481	20263	3043
strassen_base_unrolled	128	29756	557189	117958	11309	29569	2529
blocked	256	530718	2184201	416775	77170	48608	19022
blocked_unrolled	256	54517	1920646	400536	21469	31497	7574
strassen	256	744881	4819487	926840	195928	729359	74040
strassen_base	256	583778	3910705	985563	143832	81636	32558
strassen_base_unrolled	256	543989	3413127	832092	173829	325129	25300
blocked	512	1952469	50424533	6319795	283308	243829	107298
blocked_unrolled	512	1789570	50362059	6928213	278964	161390	118632
strassen	512	5998787	39098267	7550786	2248439	4928645	483031
strassen_base	512	4058780	24337397	5273181	1359662	2360799	469847
strassen_base_unrolled	512	3316085	23298294	4966828	1404551	3260227	400065
blocked	1024	13584384	420580929	48458008	2311313	1022948	1001670
blocked_unrolled	1024	12975540	409860992	50538516	2732012	1157647	839802
strassen	1024	53566250	287179408	55415201	19904869	39826374	4925302
strassen_base	1024	39111745	183230905	38726442	13488994	24183792	4650492
strassen_base_unrolled	1024	39396473	186233435	39232744	13161840	24589156	3847664
blocked	2048	1421242210	3780077111	382483325	120790776	15099346	2677953
blocked_unrolled	2048	1386511523	3719862326	433090663	133037183	4487872	2371289
strassen	2048	466350050	2029949389	371932349	129395734	285958089	38059769
strassen_base	2048	375032878	1360977976	291737702	118736922	178251093	35948624
strassen_base_unrolled	2048	372297105	1438895435	317727985	114261404	173813069	35849965

TABLE XIV
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (4 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000144	0.000085	0.006613	0.000157	0.000098
128	0.000808	0.000392	0.046459	0.001849	0.000790
256	0.006294	0.003605	0.095054	0.005025	0.002812
512	0.050187	0.038431	0.637927	0.029248	0.014956
1024	0.401601	0.307909	4.236705	0.201313	0.100139
2048	4.931426	2.709996	31.076542	1.419409	0.705134

TABLE XV
CACHE AND MEMORY METRICS (6 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	106808	316394	74294	12852	45807	12876
blocked_unrolled	64	17833	266823	95518	30238	27847	1451
strassen	64	32902	373714	105267	17288	15194	2627
strassen_base	64	3237	221858	83721	10844	6257	619
strassen_base_unrolled	64	11012	357242	89043	11965	3974	638
blocked	128	354007	986815	140176	56282	46465	11409
blocked_unrolled	128	294195	871234	182502	67193	26358	7841
strassen	128	112209	1104475	195513	23588	73591	17032
strassen_base	128	22011	584520	140212	8960	20528	2111
strassen_base_unrolled	128	29800	613744	152599	11298	16317	887
blocked	256	556564	2337052	464762	111843	38343	14316
blocked_unrolled	256	580607	2295366	454478	114590	51192	19672
strassen	256	771227	5039834	978141	279323	750517	70488
strassen_base	256	536417	3563842	796592	188668	406443	26467
strassen_base_unrolled	256	539201	3315502	774933	179472	413498	25539
blocked	512	2391361	50602362	5275667	416168	221730	77479
blocked_unrolled	512	2156985	52103550	5668862	428576	174984	96199
strassen	512	6576336	37510296	7046214	2177129	4713526	682693
strassen_base	512	3959854	23633186	4727227	1480725	3087372	431712
strassen_base_unrolled	512	3860008	23418219	5069587	1491102	2943425	460699
blocked	1024	15563397	428098975	59137122	2492416	900390	678409
blocked_unrolled	1024	14895681	403901048	41472083	2789340	894515	606918
strassen	1024	60627424	287535980	52746827	20374775	38334970	5386921
strassen_base	1024	45342662	184723408	38317328	15756198	23671573	5185327
strassen_base_unrolled	1024	44657571	192886746	39881899	15160114	22637182	4588779
blocked	2048	2201115820	3681957192	374321104	205187363	13088584	2834979
blocked_unrolled	2048	2042526730	3609931570	423876624	204224275	4220046	2207829
strassen	2048	507001247	2045279835	383732370	163119272	279341916	41818718
strassen_base	2048	421961455	1357934722	283726347	138405307	173921336	40433227
strassen_base_unrolled	2048	428461082	1365189956	298394555	138375100	169948814	40762209

TABLE XVI
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (6 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000109	0.000068	0.006764	0.000156	0.000098
128	0.000580	0.000295	0.046534	0.001837	0.000774
256	0.004229	0.002482	0.093403	0.004707	0.002500
512	0.033423	0.025693	0.420762	0.029062	0.013982
1024	0.268159	0.204979	2.717431	0.200315	0.098943
2048	3.297403	1.805998	19.521216	1.413392	0.694611

TABLE XVII
CACHE AND MEMORY METRICS (7 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	76073	412254	98889	31357	49009	5964
blocked_unrolled	64	37969	226986	87628	38739	36118	2195
strassen	64	24733	348379	106389	14122	16414	3322
strassen_base	64	12667	368357	96379	10204	3280	697
strassen_base_unrolled	64	4095	174207	81923	11498	9206	494
blocked	128	284501	1060366	159071	49281	49092	12985
blocked_unrolled	128	369893	1136842	200287	59376	28384	9869
strassen	128	62356	823537	177957	28055	90226	11732
strassen_base	128	14753	488832	137525	12596	20690	2456
strassen_base_unrolled	128	47610	621272	152415	13824	15627	3091
blocked	256	506902	2210501	411621	106622	53725	18842
blocked_unrolled	256	635919	2436147	450017	115661	48717	21354
strassen	256	994483	5262768	940189	283252	700331	62556
strassen_base	256	777940	3438936	846601	212393	352979	29607
strassen_base_unrolled	256	722929	3670779	845548	303021	310354	39878
blocked	512	1922506	53576949	6038959	323157	181227	115020
blocked_unrolled	512	1667019	53805432	6494121	392381	172797	80802
strassen	512	6393357	39308948	7111382	1969857	4935336	530403
strassen_base	512	4795291	25340898	4746606	1384804	3502454	446379
strassen_base_unrolled	512	4244796	22271055	4139488	1406379	4444043	405034
blocked	1024	13618742	430208673	46681076	2442513	1054545	736187
blocked_unrolled	1024	13465459	440675572	51112796	2762294	967305	682845
strassen	1024	59806528	283558616	52594844	19978820	39408699	5498849
strassen_base	1024	49256488	184236432	35047022	16642947	23954544	5622778
strassen_base_unrolled	1024	47982676	182552544	37555328	17023449	22647272	5273258
blocked	2048	2349979489	3776178733	382827271	215290946	15405557	2747916
blocked_unrolled	2048	2262928804	3722190463	439364161	230440676	4768273	2258360
strassen	2048	533865439	2054912275	384972201	167116704	279259815	44210063
strassen_base	2048	467371442	1361490084	275732868	148984967	170547383	43801070
strassen_base_unrolled	2048	473241929	1359513899	279551302	149182743	171445740	46732863

TABLE XVIII
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (7 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000116	0.000064	0.006735	0.000156	0.000097
128	0.000512	0.000273	0.046502	0.001829	0.000778
256	0.003606	0.002138	0.048200	0.003238	0.002233
512	0.028685	0.022098	0.332800	0.016993	0.010718
1024	0.228958	0.175858	2.326509	0.110655	0.060607
2048	2.841755	1.552101	17.020780	0.765089	0.412015

TABLE XIX
CACHE AND MEMORY METRICS (8 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	123997	417466	89579	27273	62451	11769
blocked_unrolled	64	42770	350586	106430	41693	33459	1989
strassen	64	31460	395795	103317	28769	36684	6845
strassen_base	64	31243	372351	97535	17429	3588	890
strassen_base_unrolled	64	59987	422358	97574	19449	2613	290
blocked	128	426925	1076248	169354	65721	62329	19150
blocked_unrolled	128	337437	973809	197966	74477	41128	9926
strassen	128	98601	1156740	216990	25550	65085	13458
strassen_base	128	43895	599659	150674	10573	20100	3238
strassen_base_unrolled	128	22141	507632	142174	10366	31912	2594
blocked	256	456862	2379330	459940	53076	35228	23731
blocked_unrolled	256	440824	2564176	508957	62743	37303	20451
strassen	256	1090224	6047162	1066332	267070	594141	104079
strassen_base	256	742426	3473610	852803	209507	352666	36377
strassen_base_unrolled	256	602210	3325383	849429	286360	402460	36417
blocked	512	1849395	53338400	6157514	209610	199685	156298
blocked_unrolled	512	1856131	54368613	6354543	242348	192833	108826
strassen	512	7278734	39988532	7206651	2047343	4834524	804366
strassen_base	512	3783270	23822622	5081414	1593535	3026468	297281
strassen_base_unrolled	512	4748731	26118076	4782958	1302240	3748685	392181
blocked	1024	12810809	425893728	47110708	2319432	1331904	824716
blocked_unrolled	1024	13095814	429707111	47331569	2694376	1308145	970168
strassen	1024	65765327	291055944	53504297	21844239	37245374	5789137
strassen_base	1024	51065660	185950310	36158152	16519313	23071546	5849164
strassen_base_unrolled	1024	54353683	189830970	39842122	17013500	20004729	4935133
blocked	2048	919069887	3786019032	380242239	71787691	15251774	2398237
blocked_unrolled	2048	966416129	3727339731	438266491	94198389	4770150	2442967
strassen	2048	550579309	2070815936	397273449	180233913	281884795	43630231
strassen_base	2048	465901549	1361893791	273297937	148904409	175023464	45879417
strassen_base_unrolled	2048	461608358	1364111051	278915690	148847470	175188588	46393645

TABLE XX
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (8 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000082	0.000057	0.006664	0.000156	0.000098
128	0.000473	0.000238	0.046629	0.001823	0.000788
256	0.003134	0.001856	0.048166	0.003259	0.002184
512	0.025109	0.019290	0.333185	0.018045	0.010872
1024	0.202970	0.154077	2.170999	0.113219	0.060976
2048	2.465088	1.344505	15.903518	0.760444	0.413620

TABLE XXI
CACHE AND MEMORY METRICS (16 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	112919	388813	101291	30544	81406	14592
blocked_unrolled	64	31464	305190	107449	45054	45692	2882
strassen	64	31756	379935	136150	30464	28299	4949
strassen_base	64	14895	237874	110826	20093	5636	707
strassen_base_unrolled	64	11332	217529	95467	15448	10846	684
blocked	128	707630	1719533	265826	70888	78202	34126
blocked_unrolled	128	558893	1481813	245017	66196	90452	27015
strassen	128	112459	1167199	240748	42014	87372	13104
strassen_base	128	20121	536968	157125	17524	25940	3807
strassen_base_unrolled	128	26304	638353	162122	11895	26573	2153
blocked	256	1029339	3844615	601537	115782	85526	36668
blocked_unrolled	256	1313153	4067581	612749	210902	98122	36711
strassen	256	1225452	6486279	1325708	354148	404657	126495
strassen_base	256	650124	3687010	966340	223509	359920	41661
strassen_base_unrolled	256	706233	3765025	1006577	237893	304060	49347
blocked	512	1837917	51660693	6402296	217535	209660	103896
blocked_unrolled	512	2078425	53347747	6747129	227819	259112	166822
strassen	512	8433658	41333158	7844963	2938906	4483914	819601
strassen_base	512	4094939	25728924	5145750	1293522	2623469	611350
strassen_base_unrolled	512	4869391	25321294	5268328	1459120	3231707	618945
blocked	1024	14363161	409041849	46666182	1881497	1204030	992639
blocked_unrolled	1024	12990239	422183750	50117840	2727024	1312727	993099
strassen	1024	80120207	292910653	52661030	24290421	36965878	7488498
strassen_base	1024	59407062	206825432	55199444	19429238	22650969	6367801
strassen_base_unrolled	1024	59533959	187031770	39351769	18534695	21439260	6237390
blocked	2048	565403731	3792147922	378479673	44438640	15675656	2522684
blocked_unrolled	2048	554324431	3750716616	434965077	61783014	4749134	2516887
strassen	2048	638002310	2074673081	381932390	202257035	276338662	54324804
strassen_base	2048	494804429	1369721534	278723123	152282158	173593185	47229570
strassen_base_unrolled	2048	499051166	1363461125	285276296	155264569	171392386	49802890

TABLE XXII
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (16 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000089	0.000057	0.006769	0.000156	0.000098
128	0.000280	0.000146	0.046664	0.001803	0.000791
256	0.001613	0.001003	0.048363	0.003352	0.002370
512	0.012609	0.009674	0.192302	0.018415	0.011811
1024	0.101205	0.077406	1.122448	0.113072	0.062539
2048	1.233346	0.671075	7.877060	0.776766	0.414734

TABLE XXIII
CACHE AND MEMORY METRICS (32 THREADS)

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	133617	637731	194191	44872	45157	4081
blocked_unrolled	64	116153	493748	134644	26805	81028	21753
strassen	64	88161	613494	222051	43095	39148	9639
strassen_base	64	28758	399119	133159	23696	7194	1079
strassen_base_unrolled	64	72338	464625	140419	25020	4202	1177
blocked	128	757458	2302871	327882	81970	142877	54836
blocked_unrolled	128	605812	1917788	368370	75985	124317	37533
strassen	128	143154	1035819	214360	48640	154950	32126
strassen_base	128	32583	643953	187934	24638	25616	3259
strassen_base_unrolled	128	58837	710182	191691	28677	27157	3346
blocked	256	2912394	7109618	814923	234152	240515	114419
blocked_unrolled	256	3269170	7740455	848163	263617	224818	96530
strassen	256	1001953	5953370	1245361	359069	659759	113840
strassen_base	256	1085879	4735816	1246423	311714	445383	86098
strassen_base_unrolled	256	885773	4568395	1295466	335542	428298	75083
blocked	512	3215976	52312129	6611099	546754	272910	120614
blocked_unrolled	512	2891392	55995716	7433134	986575	261723	148507
strassen	512	10436674	41653311	7526906	3170178	4613869	982854
strassen_base	512	4847449	25991466	5600823	1571069	2708591	830249
strassen_base_unrolled	512	4782801	26047125	5806396	1460618	2872772	803040
blocked	1024	13893051	415508580	47157936	1996309	1686352	1086164
blocked_unrolled	1024	12977803	442911296	52943299	2338482	1632697	1209902
strassen	1024	91656575	305646557	54207702	27493738	37532900	8014760
strassen_base	1024	57251562	188043512	37166238	18386755	21171503	4982059
strassen_base_unrolled	1024	53438005	189529114	38929193	17716213	23371696	4777918
blocked	2048	391372652	3812568726	380735407	32666099	16266814	3058436
blocked_unrolled	2048	435320109	3766641790	436473166	55427315	5362465	2717823
strassen	2048	762639980	2157194840	401233660	239565001	275392684	59209975
strassen_base	2048	485875207	1514538496	396042037	155568051	165460572	43937080
strassen_base_unrolled	2048	489597099	1354810948	281918882	153568362	172557182	47418581

TABLE XXIV
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (32 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000089	0.000063	0.007781	0.000200	0.000144
128	0.000210	0.000116	0.046935	0.001804	0.000879
256	0.001566	0.000954	0.048535	0.003750	0.002690
512	0.008232	0.007884	0.101438	0.020137	0.013206
1024	0.059141	0.039281	0.613087	0.117994	0.061904
2048	0.633683	0.447845	3.975442	0.773871	0.415209

TABLE XXV
CACHE AND MEMORY METRICS FOR 64-THREAD IMPLEMENTATION

Algorithm	Size	Cache Misses	Cache References	LLC Loads	LLC Load Misses	LLC Stores	LLC Store Misses
blocked	64	435568	1211912	263163	40210	60542	38880
blocked_unrolled	64	289836	996175	310103	44008	37447	25458
strassen	64	128534	886383	306004	55320	79437	15184
strassen_base	64	16465	258958	152309	27546	40119	6252
strassen_base_unrolled	64	35500	287652	128440	23484	55203	6396
blocked	128	890273	3376121	459269	94483	431391	144263
blocked_unrolled	128	492761	2171493	504583	177319	207314	35730
strassen	128	190386	1706264	514600	93957	104333	19247
strassen_base	128	19732	505865	199436	32340	48769	3860
strassen_base_unrolled	128	92662	734125	191113	21692	74310	25926
blocked	256	3638962	10633922	1102206	312917	516461	238164
blocked_unrolled	256	3032017	9338944	1120908	297897	506347	143988
strassen	256	1395075	7218785	1470351	385068	778984	184652
strassen_base	256	1325717	6140550	1896259	452099	559677	116659
strassen_base_unrolled	256	1264104	5984767	1884976	440223	619241	121873
blocked	512	3624192	41602807	5501741	343096	372452	222019
blocked_unrolled	512	6128861	51386083	5808971	527877	821246	164610
strassen	512	12981992	58113818	9181457	3615708	6149170	1215403
strassen_base	512	5319917	28688357	6425104	1697627	3249025	741438
strassen_base_unrolled	512	5408047	29238216	6292378	1570340	3797098	906329
blocked	1024	23647278	350593614	46785205	2578011	2441305	2409795
blocked_unrolled	1024	27349624	413186918	60486573	3929077	3264420	3331859
strassen	1024	120058399	411744292	63483751	31992587	41069659	8483138
strassen_base	1024	53264609	191102853	38092484	18667141	22175390	4589799
strassen_base_unrolled	1024	56474692	196541749	40018798	18428025	19768983	4852289
blocked	2048	831021872	6206413377	1161637493	71938988	48550802	6844392
blocked_unrolled	2048	908395022	4934031847	911316049	121805245	11076625	4039155
strassen	2048	984940562	2864828655	437871663	251202759	287739371	63965825
strassen_base	2048	466255646	1387540210	293522493	153611943	164435982	36479315
strassen_base_unrolled	2048	477469328	1393666205	290797116	153925695	164437595	40540592

TABLE XXVI
AVERAGE EXECUTION TIME FOR PARALLEL IMPLEMENTATIONS (64 THREADS)

Size	Blocked	Blocked Unrolled	Strassen Naive	Strassen Base	Strassen Base Unrolled
64	0.000109	0.000076	0.008205	0.000208	0.000153
128	0.000159	0.000125	0.048614	0.001795	0.000875
256	0.000840	0.000563	0.073043	0.004308	0.003420
512	0.006330	0.004546	0.100938	0.024958	0.017468
1024	0.050701	0.037105	0.575866	0.133473	0.079070
2048	0.453593	0.312003	3.646965	0.783679	0.430842

TABLE XXVII
AVX IMPLEMENTATION ELAPSED TIMES (SECONDS) FOR BLOCKED VS. STRASSEN'S ALGORITHMS

Matrix size	Threads	Blocked (s)	Strassen (s)
64×64	8	0.000029	0.000040
64×64	16	0.000036	0.000036
64×64	32	0.000045	0.000042
64×64	64	0.000067	0.000054
128×128	8	0.000069	0.000270
128×128	16	0.000062	0.000253
128×128	32	0.000073	0.000266
128×128	64	0.000110	0.000267
256×256	8	0.000254	0.001396
256×256	16	0.000188	0.001343
256×256	32	0.000176	0.001437
256×256	64	0.000199	0.001496
512×512	8	0.001418	0.006845
512×512	16	0.000905	0.006233
512×512	32	0.000707	0.005859
512×512	64	0.000714	0.007387
1024×1024	8	0.009724	0.037926
1024×1024	16	0.005506	0.030180
1024×1024	32	0.003531	0.025490
1024×1024	64	0.003625	0.030282
2048×2048	8	0.085875	0.185182
2048×2048	16	0.046733	0.128535
2048×2048	32	0.027950	0.098014
2048×2048	64	0.026438	0.100100
4096×4096	8	0.688306	1.176284
4096×4096	16	0.362733	0.717436
4096×4096	32	0.198498	0.515005
4096×4096	64	0.183710	0.539249
8192×8192	8	5.374215	8.119315
8192×8192	16	2.760151	4.701193
8192×8192	32	1.500172	3.077888
8192×8192	64	1.264944	3.224309
16384×16384	8	42.238991	53.022141
16384×16384	16	21.149659	30.830721
16384×16384	32	12.401352	19.338095
16384×16384	64	9.816138	19.648965