ECTI Transactions on Computer and Information Technology
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

Power Efficient Strassen's Algorithm using AVX512 and OpenMP in a Multi-core Architecture

Nwe Zin Oo[1] and Panyayot Chaikan[2]

ABSTRACT

This paper presents an effective implementation of Strassen's algorithm for matrix-matrix multiplication on shared memory multi-core architecture. The proposed algorithm aims to augment the computation speed in terms of *GFLOPS* performance on average **4.5** and **4.1** times faster than *Eigen* and *OpenBLAS*, respectively while reducing the power consumption to as low as possible. Our algorithm relies on using AVX512 intrinsics, loop unrolling factor, and OpenMP directives. A new 2D blocking data allocation pattern is proposed for Strassen's algorithm to provide optimized cache temporal and spatial locality. The proposed implementation reduced not only the amount of main memory but also the burden of unnecessary memory allocation/deallocation and data transferring for each level of recursion in Strassen's algorithm. Moreover, the proposed algorithm consumed, on average, **4.25** and **3.67** times lower energy than the multiplication functions of the *Eigen* and *OpenBLAS* libraries, respectively. To measure the computational performance with the awareness of power consumption, *GFLOPS per Watt (GFPW)* is calculated, which outperformed on average **3.78** and **3.47** times higher than those of Eigen and OpenBLAS libraries, respectively.

Article information:

Keywords: AVX, OpenMP, Strassen's algorithm, 2D blocking, Energy consumption, Power efficiency, Shared memory

Article history:
Received: April 28, 2022
Revised: July 14, 2022
Accepted: January 6, 2023
Published: January 28, 2023 (Online)

DOI: 10.37936/ecti-cit.2023171.248320

1. INTRODUCTION

Modern computers are fitted with high-performance multi-core processors which can execute intensive mathematical applications with parallel programs to obtain maximum computation speed. However, the energy consumption of the system is increasing, while implementing multiple threads in a parallel version of intensive computing. Therefore, energy-efficient and power-aware computing is becoming an important issue when designing scientific and engineering software that requires many computations. For example, [1] proposed a method to minimize the number of data read per floating-point operation, while [2] utilized *Dynamic Voltage Frequency Scaling (DVFS)*. Work by [3] demonstrates that using AVX instructions could save energy on the load/store operations by reducing the amount of data transfer from memory.

Since matrix-matrix multiplication is a significant operation in scientific and engineering applications, not only does an intensive computational time but also high energy consumption is required to complete a process. Strassen's algorithm is an efficient method for improving the performance of matrix-matrix multiplication in [4] by utilizing both the shared memory and thread mechanisms on GPUs to fuse additional operations and avoid extra workspace for one-level and two-levels Strassen's algorithm. Several methods [5-9] have been proposed to improve its performance. In this paper, an efficient implementation of saving energy consumption for Strassen's algorithm is proposed by utilizing AVX-512 and OpenMP. Unlike the method proposed in [4], which uses both the memory and thread hierarchies on 5-cores and 10-cores GPUs to avoid extra workspace, and [10], which aimed for one-level recursion parallel Strassen's algorithm on a dual-core processor, our algorithm focuses on different recursion levels for power-efficient Strassen's algorithm on multi-core architecture testing on the CPUs and it boosts achieving high computational speed with the awareness of energy saving.

[1,2] The authors are with Department of Computer Engineering, Faculty of Engineering Prince of Songkla University, Hat Yai, Thailand., E-mail: 5910130015@email.psu.ac.th and panyayot.c@psu.ac.th

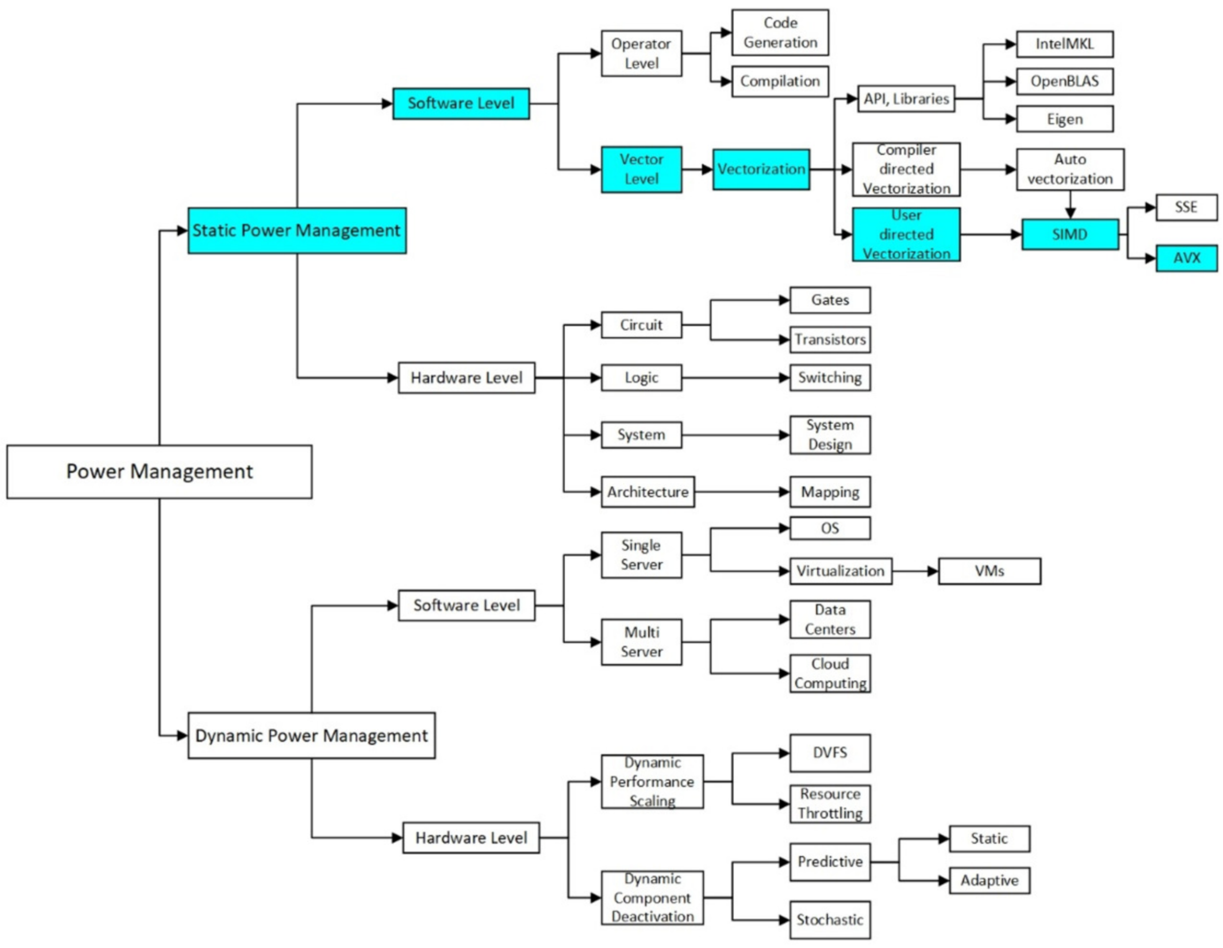


Fig.1: *Power management techniques.*

This article has organized as follows. In section 2, the previous related works are discussed, and the details of power management techniques are presented in section 3. In section 4, an optimized 2D blocking pattern has been proposed, and power-efficient Strassen's algorithm is implemented in section 5. Section 6 explains the experimental results, while sections 7 and 8 are dedicated to the discussion and conclusion, respectively.

2. RELATED WORKS

The energy efficiency of a program calculation has become one of the most interesting research areas in recent years due to the high-power consumption on high-performance computing systems or the battery usage time on embedded computer systems. For this reason, energy and power consumption are essential constraints for optimizing programs running on shared and distributed memory architecture. For example, the *DVFS* technique has been utilized in computing clusters [11]. The voltage-lowering technique has been proposed for the GPU-based system to reduce power consumption [12]. Energy consumption measuring models were proposed using the *Running Average Power Limit (RAPL)* in the Linux kernel [13]. The power management architecture utilizing extra hardware measurement equipment for Intel's Sandy bridge has been proposed in [14]. *Jakobs et al.* suggested that using vectorization can reduce the system's power consumption [15].

3. TWO TYPES OF POWER MANAGEMENT

Power consumption is the energy used per unit of time [16]. In other words, power can define as the rate at which energy is consumed. The unit of power in watts (W), which is Joules per second. Most digital systems consume both dynamic and static power while running an application. Moreover, heat generation depends on the amount of power consumption. A kilowatt-per-hour (kWh) is the amount of energy equivalent to a power of one thousand watts (*1000 W*) running for one hour [16]. The relation of power and energy consumption, while the system performs the application workloads can be defined as $P = E/t$, where P is power, E is energy, and t is the execution time.

Decreasing power consumption does not always reduce the energy consumed by a program [16]. For instance, power consumption can be reduced by lowering the processor's performance, and the program might take a longer execution time to complete its goal but consumes the same or more amount of energy. Therefore, a high-performance, the energy-efficient system should consume low power and take low computation time. Power consumption can be reduced using *Dynamic Power Management (DPM)* or applying *Static Power Management (SPM)* [16]. Both techniques have different software and hardware-level management to accomplish the requirements of the intended efficient power management system. Fig. 1 illustrates the details of power management techniques for modern computer architecture.

There are two types of power consumption: static and dynamic [17]. Static power is the power consumed when there is no circuit activity. On the other hand, dynamic power is the consumed power while the inputs are active [18]. There are many possible solutions to minimize power consumption in a $CMOS$ system depending on the characteristics of the techniques in which the processor is fabricated. Among them, $DVFS$ is one of the optimized methods for DPM. $DVFS$ controllers were presented and can simultaneously adjust the voltage and clock speed based on a command set [11]. The voltage and frequency of memory can be adjusted based on memory bandwidth utilization using $DVFS$ [19]. In addition, dynamic power consumption can be significantly decreased by reducing the power supply voltage to the defined level that provides the required performance as a dynamic performance scaling at the hardware level. Some techniques in [20][21] energy-efficient dynamic *Virtual Machines (VM)* consolidation are applied at the single-server or multi-server level for the distributed system.

In SPM, techniques on the software and hardware level are different from each other. From a hardware point of view, the voltage drops on diodes, logical gates, transistors, and switching devices are the main keys to reducing static power consumption. From software level perception, the reduction of an arithmetic operator and vectorization are the main keys to building a software design that can dramatically adjust the performance and power consumption of the system. The use of an optimized compilation process and vectorization can provide energy efficiency when a system supports *Single-Instruction Multiple-Data (SIMD)* instruction set, such as *Advanced Vector Extensions (AVX)* [22]. Today's modern compilers can utilize these instructions by applying auto-vectorization or manual-vectorization techniques or by choosing software libraries that optimize for vector instruction, such as *Intel Math Kernel Library (MKL)*, *OpenBLAS* [33], and *Eigen* [32]. In this paper, we proposed techniques for reducing the power consumption of matrix-matrix multiplication. Our proposed methods only focus on static power reduction, as highlighted in blue in Fig. 1. Our approach works solely with software without needing extra hardware.

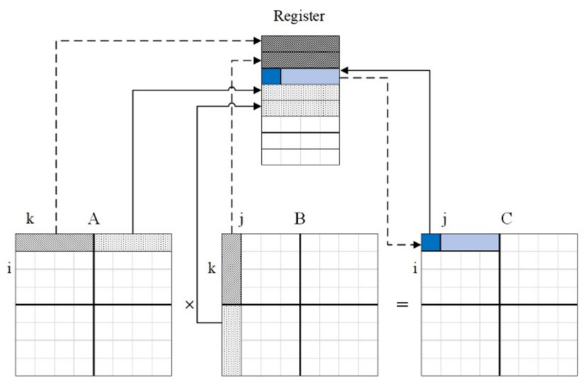


Fig.2: *A conventional block-wised matrix-matrix multiplication.*

4. PROPOSED 2D BLOCKING MULTIPLICATION USING AVX AND OPENMP

The proposed matrix multiplication performs parallel execution by observing for energy consumption of Strassen's algorithm at function-level and loop level over time. To calculate the power consumption of cache utilization, the number of cache misses and hits rate of the memory accesses are mainly dependent due to the microarchitecture implementation of the memory hierarchy. The goal of power efficiency is to maximize performance with a given less energy consumption to fulfill the multiplication process. Theoretically, when increasing the number of processor cores, the execution time required at each core can be decreased, which leads to improved performance. From a computational perspective point of view, power consumption and performance of shared memory accesses and computation can be varied due to different parallel algorithms. For this reason, parallel performance and energy cost depend not only on the number of cores but also on the implementation of the parallel algorithm [23].

Suppose A and B are square matrices of size $N \times N$. A product $C_{i,j}$ is obtained by performing $A_{i,k} \times B_{k,j}$, as shown in Fig. 2. Many iterations are required to obtain all the product elements. Typically, in sequential square matrix multiplication, there are $2n^3$ loads for reading data of matrices A and B and $2n^2$ loads/stores for a reading and writing the data to matrix C.

When the block-wise pattern is applied to matrix-matrix multiplication, the amount of main memory access is reduced. The matrices A, B, and C are divided into sub-blocks of size $b \times b$, which can be kept inside the faster memory such as cache. However, keeping all these three sub-blocks in the cache simultaneously is difficult due to the cache size limitation. This reason motivates us to implement an efficient al-

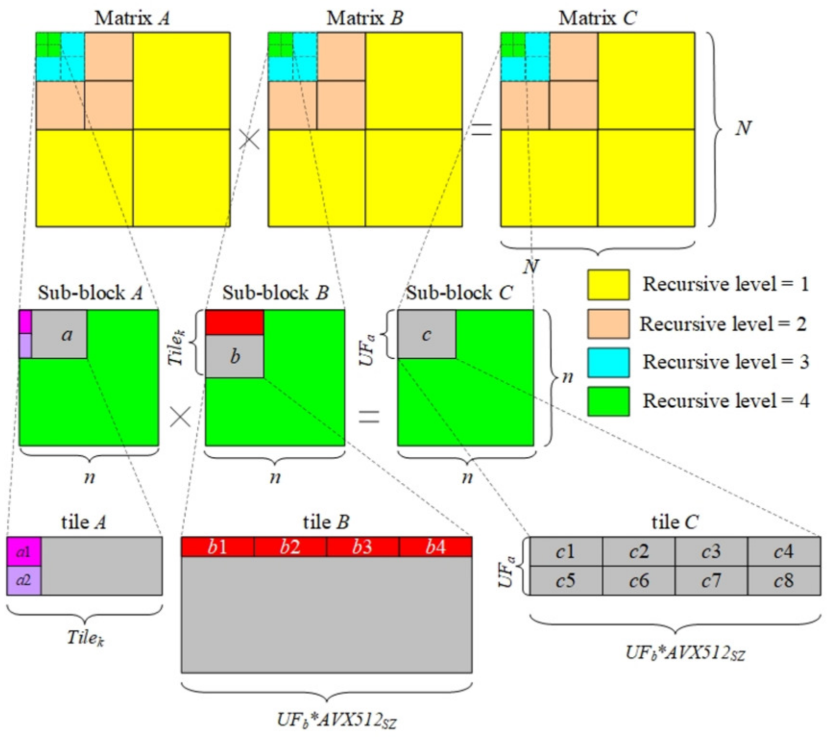
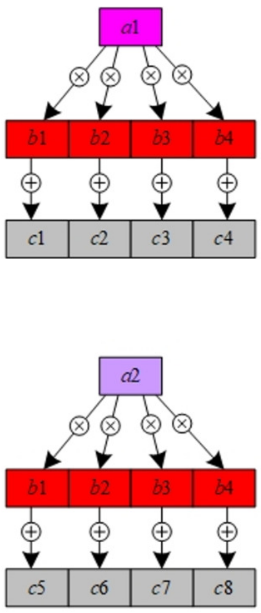


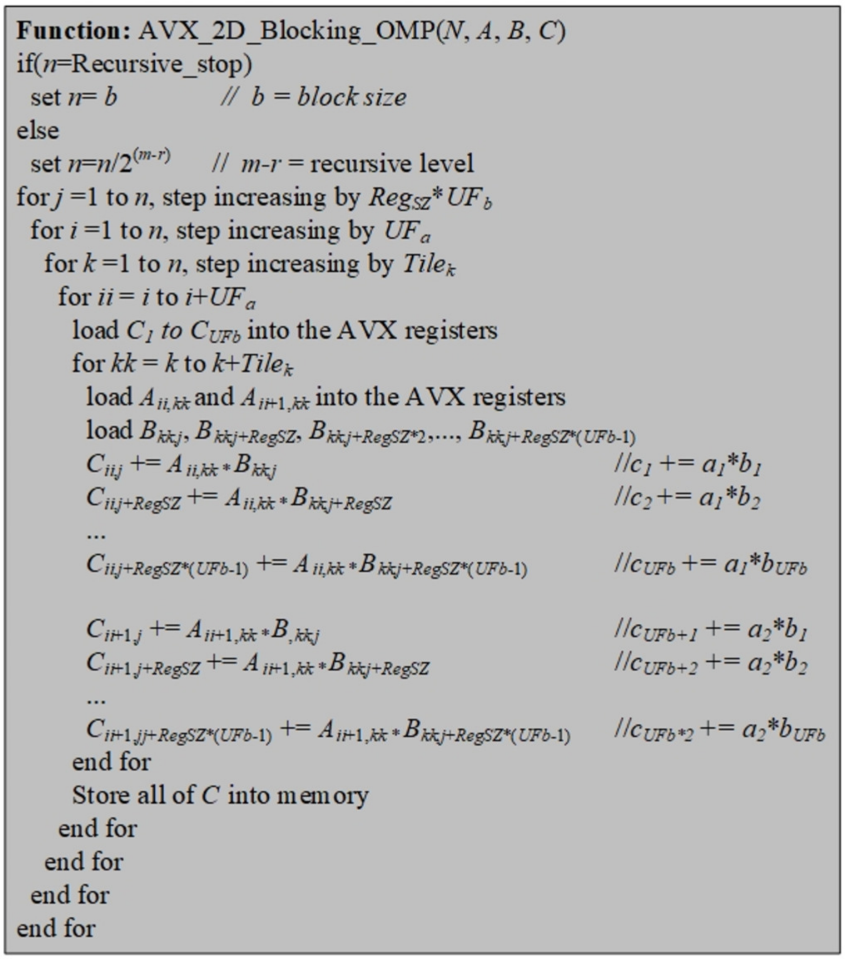
Fig.3: *The proposed 2D blocking data allocation pattern for power-efficient Strassen's algorithm.*

location pattern for Strassen's matrix-matrix multiplication. In modern computer architecture, the performance is improved when the data allocation pattern is matched with the storage layout [24]. Even though the AVX relaxes the memory alignment access requirement [25], we keep the matrices A, B, and C in the 32-bit aligned address to maximize the efficiency of memory loads and stores.

Fig. 3 demonstrates the new optimized data allocation pattern for Strassen's algorithm. Although Strassen's recursive approach is based on the divide and conquers method, which can be set to any level of recursion, we decided to stop the recursive call when the total size of the 2D sub-blocks from matrices A, B, and C fits inside the cache to reduce its overhead of memory allocation/deallocation. Let the sub-block at the last recursive level be of size $n \times n$; UF_a elements from sub-block A are fetched from memory and then broadcasted to $AVX512SZ$ elements inside the AVX-512 registers. Each line of the tile of the data from matrix B is fetched $UF_b * AVX512SZ$ elements at a time before being multiplied with the broadcasted data from A using a *fused-multiply-accumulate (FMA)* instruction, producing $UF_b * AVX512SZ$ results of matrix C, as shown in Fig. 3. The AVX512SZ parameter is set to **8** and **16** for double and single-precision floating-point data, respectively. When matrix multiplication is applied with AVX intrinsics, not only the number of multiplications and additions operations are reduced but also the number of loads/stores is reduced too. Since AVX512 intrinsic instructions can load eight double-precision from matrix A and matrix B and store eight multiplied results in matrix C with only a single operation, the minimum tile size can be defined with the $AVX512SZ$ value.

The proposed algorithm supports multi-threading via the OpenMP directive calls, the ***#pragma omp parallel for***. Each thread can process the multiplication operation and store the partially multiplied results into the corresponding indexes (ii, j) of matrix C. The main advantage of our contribution is enhancing the spatial and temporal localities by consecutive memory addresses for the sub-block of matrices A, B, and C as much as possible at each block-wise multiplication. While $a1$ is multiplied with packed data from $b1$, $b2$, $b3$, and $b4$, $a2$ is multiplied in parallel with the same packed data from $b1$, $b2$, $b3$, and $b4$; therefore the number of data accessed from slow memory can be reduced, due to these have already existed in the cache. This method leads to an increase in the number of cache hits, especially for matrix B, and reduces the power consumption for data searching, transferring, and accessing to/from the main memory. Moreover, the proposed pattern is optimized for data and task parallelism using AVX512 registers and OpenMP multi-threading. One of the factors that can improve performance is using the available registers that have been used as much as possible. The available registers in AVX-512 are **32** for integer and floating-point arithmetic operations. In Fig. 4, the optimized UFa and UFb are set to **2** and **4**, respectively. There are





```
Function: AVX_2D_Blocking_OMP($N$, $A$, $B$, $C$)
if($n$=Recursive_stop)
  set $n$= $b$          // $b$ = block size
else
  set $n$=$n/2^{(m-r)}$     // $m-r$ = recursive level
for $j$ =1 to $n$, step increasing by $Reg_{SZ}$* $UF_b$
  for $i$ =1 to $n$, step increasing by $UF_a$
    for $k$ =1 to $n$, step increasing by $Tile_k$
      for $ii$ = $i$ to $i$+$UF_a$
        load $C_1$ to $C_{UFb}$ into the AVX registers
        for $kk$ = $k$ to $k$+$Tile_k$
          load $A_{ii,kk}$ and $A_{ii+1,kk}$ into the AVX registers
          load $B_{kkj}$, $B_{kkj+RegSZ}$, $B_{kkj+RegSZ*2}$,..., $B_{kkj+RegSZ*(UFb-1)}$
          $C_{ii,j}$ += $A_{ii,kk} * B_{kkj}$                          //$c_1$ += $a_1$*$b_1$
          $C_{ii,j+RegSZ}$ += $A_{ii,kk} * B_{kkj+RegSZ}$              //$c_2$ += $a_1$*$b_2$
          ...
          $C_{ii,j+RegSZ*(UFb-1)}$ += $A_{ii,kk} * B_{kkj+RegSZ*(UFb-1)}$    //$c_{UFb}$ += $a_1$*$b_{UFb}$

          $C_{ii+1,j}$ += $A_{ii+1,kk} * B_{,kkj}$                     //$c_{UFb+1}$ += $a_2$*$b_1$
          $C_{ii+1,j+RegSZ}$ += $A_{ii+1,kk} * B_{kkj+RegSZ}$          //$c_{UFb+2}$ += $a_2$*$b_2$
          ...
          $C_{ii+1,jj+RegSZ*(UFb-1)}$ += $A_{ii+1,kk} * B_{kkj+RegSZ*(UFb-1)}$   //$c_{UFb*2}$ += $a_2$*$b_{UFb}$
        end for
        Store all of $C$ into memory
      end for
    end for
  end for
end for
```

Fig.4: *The pseudo-code of our proposed 2D blocking multiplication using AVX.*

two registers for loading the data elements from matrix A ($a1$ and $a2$), **four** registers for loading the data elements from matrix B ($b1$, $b2$, $b3$, $b4$), eight registers for temporarily maintaining the partial multiplied results ($r1$, $r2$, $r3$, $r4$, $r5$, $r6$, $r7$, $r8$), and **eight** registers for storing the complete multiplication results in matrix C. In this case, there are **22** registers used in parallel matrix-matrix multiplication.

We found that the proposed loop transformation and data allocation give optimized results and achieve the best parallelism for square matrix-matrix multiplication when using AVX512 intrinsic instructions with Strassen's algorithm for the matrix size, which is the power of two (N^2). Finally, when applying a proposed ***AVX_2D_Blocking_OMP( )*** function in Strassen's algorithm, the optimized performance matrix in terms of power efficiency and improved $GFLOPS$ can be obtained on a multi-core architecture.

There are two kinds of arithmetic operations in Strassen's algorithm: matrix multiplications and matrix additions/subtractions, as shown in Fig. 5 (b). The intermediate results must be stored in temporary locations when recursive functions are called in them. When sub-matrices $M1$ to $M7$ are calculated on shared memory, **15** sub-matrices are required for computing **seven** multiplications and **ten** additions/subtractions. Moreover, temporary $C11$, $C12$, $C21$, and $C22$ memory allocations need to calculate **eight** additions/subtractions in Strassen's algorithm. In a square matrix-matrix multiplication, although matrices A, B, and C require $3N^2$ elements of memory, however, our method utilizing 2D blocking requires $3(N/2)^2$ elements of memory. Therefore, if only one level of Strassen's algorithm is applied to our 2D blocking, total memory usage is $3(N/2)^2 + 15(N/2)^2 + 4(N/2)^2 = 22(N/2)^2$ elements. We can calculate the total amount of memory usage by using the following:

Table 1: *The complexity of sequential and Strassen's matrix-matrix multiplication algorithms.*

Matrix-Matrix Multiplication Algorithm	Number of Operations	Complexity
Sequential	$N^2(N + (N - 1))$	$O(N^3)$
Strassen's algorithm	$N^{\log_2(7/8)} 2N^3$	$O(N^{2.8074})$

$$Mem_{Total} = 3\left(\tfrac{N}{2}\right)^2 + \textstyle\sum_{i=0}^{m-r-1} 15\left(\tfrac{N}{2^{(i+1)}}\right)^2 + 4\left(\tfrac{N}{2^{(m-r)}}\right)^2 \quad (1)$$

where,

Mem_{Total} = the total amount of memory usage in Strassen's algorithm,

N = the matrix dimension of 2^m where ($m > 0$), and

$m - r$ = the number of recursive levels when the block size is $b = 2^r$ where ($r \geq 0$).

5. IMPLEMENTATION OF POWER-EFFICIENT STRASSEN'S ALGORITHM

Parallel computing comprises a process of independent subtasks and interactions between the parallel subtasks. Therefore, the performance and energy cost of parallel computing depend not only on the number of processing cores but also on the structure of that parallel algorithm. *Parallel matrix-matrix multiplication (PMMM)* on shared memory architecture is one of the challenging problems of high-performance computing applications. Strassen's algorithm is a recursive matrix multiplication and is faster than a standard multiplication algorithm which is useful in practice for large matrices multiplication. It is based on a divide and conquer strategy that recursively divides the data into the equal square size of four sub-matrices until the data size can directly processed by the processor architecture.

Standard blocking matrix-matrix multiplication needs eight multiplication and eight addition operations, as shown in Fig. 5 (a). However, Strassen's algorithm needs only **seven** multiplication and **18** addition/subtraction operations to obtain the same result, as shown in Fig. 5 (b). Therefore, it reduces the computational complexity by reducing the number of multiplication operations. Since the multiplication cost is more than that of the addition/subtraction operations, so we can use Strassen's algorithm not only for improving the computational speed but also for reducing the energy consumption in the matrix-matrix multiplication.

Table 1 shows the complexity of different multiplication algorithms. Strassen's algorithm can reduce the number of multiplications by executing the seven sub-matrices of computing ($M1$, $M2$, $M3$, $M4$, $M5$, $M6$, $M7$), as shown in Fig. 5.

Ordinary matrix-matrix multiplication needs N^3 multiplication operations and $N^2 * (N-1)$ addition operations. Its complexity is $O(N^3)$, while Strassen's algorithm requires only $7/8\log_2(N)2N^3$ arithmetic operations which is $O(N^{2.807})$ [4].
To compute the number of arithmetic operations that are theoretically required by Strassen's algorithm, let the input source matrices A and B be of size $N \times N$ ($N = 2^m$ where $m > 0$). After dividing into four equal square sub-matrix or sub-block of size $N^2/4$ at the top level, **seven** multiplications are required. Each multiplication to obtain M_i further requires seven multiplications and **18** additions/subtractions for each level of calculation.

If Strassen's algorithm will be applied to the source matrices only at the top level without any recursive call, the source matrices A and B are divided into four equal sub-blocks as shown in Fig. 3. Therefore, each sub-block multiplication requires $(N/2)^3$ multiplication operations and $(N/2)^2 * (N/2 - 1)$ addition/subtraction operations. Since Strassen's algorithm needs **7** multiplications and **18** additions/subtractions for each level of calculation, we can calculate total number multiplication operations for this case by

$$Num_{Multiplication} = 7 \times \left(\frac{N}{2}\right)^3 \quad (2)$$

The total number of addition/subtraction operations required in this case is obtained by

$$Num_{AddSub} = 18\left(\frac{N}{2}\right)^2 + 7\left(\frac{N}{2}\right)^2\left(\frac{N}{2} - 1\right) \quad (3)$$

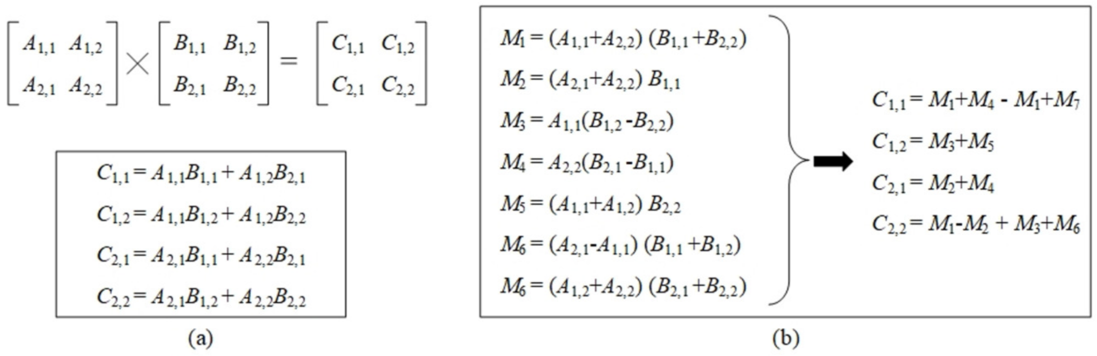
When Strassen's algorithm has recursively called down to the sub-block of size $2^r 2^r$, the total number of multiplication operations per recursive level can be calculated by using the following Eq. (4).

$$7^{(m-r)} \times \left(\frac{N}{2^{(m-r)}}\right)^3 \quad (4)$$

where r is an integer value. For example, if the matrix size at the top level is 64×64 and if $r = 2$, then the number of the recursive level becomes four ($m - r = 4$). At the first level, seven multiplications are needed for the sub-matrix of size $N^2/2^2$, which is equal to $7 \times (N/2)^3$ multiplication operations.

At the second level of recursion, 7^2 multiplications will be required for the sub-matrix of size $N^2/4^2$ which is equal to $7^2 \times (N/4)^3$ multiplication operations. At the third level, 7^3 multiplications will be needed for the sub-block of size $N^2/8^2$ which is equal to $7^3 \times (N/8)^3$ multiplication operations. At the fourth level of recursion, 7^4 multiplications are required for the submatrix of size $N^2/16^2$, which is equal to $7^4 \times (N/16)^3$.

The total number of additions/subtractions ($T\_Num_{strassen}$) required for Strassen's algorithm when it is recursively called down to the sub-block of size $2^r \times 2^r$ can be calculated by



$A_{1,1}$ $A_{1,2}$ $A_{2,1}$ $A_{2,2}$ × $B_{1,1}$ $B_{1,2}$ $B_{2,1}$ $B_{2,2}$ = $C_{1,1}$ $C_{1,2}$ $C_{2,1}$ $C_{2,2}$
$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$
(a)
$M_1 = (A_{1,1}+A_{2,2})(B_{1,1}+B_{2,2})$
$M_2 = (A_{2,1}+A_{2,2})B_{1,1}$
$M_3 = A_{1,1}(B_{1,2}-B_{2,2})$
$M_4 = A_{2,2}(B_{2,1}-B_{1,1})$
$M_5 = (A_{1,1}+A_{1,2})B_{2,2}$
$M_6 = (A_{2,1}-A_{1,1})(B_{1,1}+B_{1,2})$
$M_6 = (A_{1,2}+A_{2,2})(B_{2,1}+B_{2,2})$
$C_{1,1} = M_1+M_4 - M_1+M_7$
$C_{1,2} = M_3+M_5$
$C_{2,1} = M_2+M_4$
$C_{2,2} = M_1-M_2 + M_3+M_6$
(b)

Fig.5: *Matrix-matrix multiplication (a) using the blocking method and (b) using Strassen's algorithm.*

$$T\_Num_{strassen} = \sum_{i=0}^{m-r-1} 7^i \times 18\left(\frac{N}{2^i+1}\right)^2 + 7^{(m-r)}\left(\frac{N}{2^{(m-r)}}\right)^2\left(\frac{N}{2^{(m-r)}} - 1\right) \tag{5}$$

In the remaining part of this section, we proposed a new 2D blocking multiplication for the power-efficient Strassen's algorithm based on the following significant perspectives to improve power efficiency on a shared memory architecture.

5.1 Memory Access Pattern

Data transferring between the CPU and main memory is one of the most expensive operations. As a result, power consumption is reduced by adequately organizing instructions and data in memory or caches. The most power-efficient way is accessing from cache/registers since cache accesses are more power-efficient than main memory accesses [14]. In our proposed implementation, an efficient data allocation pattern is applied, as explained in Fig. 4.

5.2 Block-size and Cache-size

Caches are a critical component in reducing energy consumption since increasing cache hit saves the data transferring between the CPU and main memory. The size of the cache is set by the CPU manufacturer and can be varied among different machines. To increase the cache utilization as much as possible, the proposed implementation calculates the sub-block size of Strassen's algorithm based on the cache information of the CPU. Since multi-core architecture is ubiquitous, most cache architectures in modern processors are divided into three levels. The first and second levels are dedicated to each processing core, and the third is shared between all the cores. We should tradeoff Strassen's algorithm be recursively called down to the level that the total memory required for the sub-block of A, B, and C fits inside the $L2$ cache to let each $L2$ cache of each processing core be reused as much as possible and prevent the L3 contention between the processing cores. This policy enables our algorithm to run on machines with different cache sizes.

5.3 Loop Unrolling Factor

Loop unrolling can minimize the cost of loop overheads in the loop-level calculation. However, when the loop is unrolled too much, power increases due to the required number of accumulated registers exceeding the number of available registers in the memory system. The effect of unrolling is highly dependent on the unrolling factor, which is how many times the loop is unrolled in a program. In [30], the different loop unrolling patterns (U_1 and U_2) were proposed for Strassen's algorithm and as a result, U_2 reduced by approximately 30% for the energy/power consumption and 67% outperformed U_1 since the number of data movements in their U_2 is less than that of their compared U_1 method. The reason is that the innermost loop of the U_2 pattern is unrolled by four elements simultaneously for matrix A and matrix B which resulted in only the next four elements from matrix A being required at the inner loop. Therefore, their proposed unrolling factor (UF) is four for both A and B matrices. The loop unrolling method in our proposed paper is different from [30] since two types of loop unrolling factors, such as UF_a for accessing only **two** elements from matrix A and UF_b for accessing **four** elements from matrix B, are unrolled in the loop iteration at line number 5, 6, and 8 as shown in Fig. 4. As a restriction, the performance can be degraded if the unrolled programming code is increased to exceed the L1 code cache and needed to be aware not to exceed the available cache size.

There are **16** and **32** available registers for AVX256 and AVX512, respectively. Therefore, the loop unrolling factor is needed to tradeoff between the code size, the number of available registers, and the execution time of an algorithm.

5.4 AVX vectorization and OpenMP

AVX is a set of instructions for performing SIMD operations on Intel CPUs. AVX-256 registers expand 128-bit SIMD registers into 256-bit registers and include the operation of the *Fused-Multiply-Accumulate (FMA)* function [25]. Since vectorization also improves energy efficiency by reducing instruction cache penalty, AVX not only increases the computation speed and data reference locality but also reduces power consumption. AVX's FMA instruction combines multiplication and addition into a single instruction named *_mm256_fmadd_pd* which can improve both performance and accuracy. Since it rounds the result only once, while separate multiplication and addition operations have two. Most compilers can perform auto-vectorization, and complier-directed auto-vectorization has strong limitations in the analysis and code transformation phases that prevent an efficient extraction of $SIMD$ parallelism in real applications [20].

OpenMP is a multi-threading API for shared-memory parallelism using a runtime library (e.g. *omp.h*) [28]. In OpenMP, each thread executes its works within OpenMP parallel regions to improve the parallelism. Moreover, work-sharing constructs (*#prgama omp parallel sections*) can be used for splitting up loop iterations among multi-threads in the function. In Strassen's algorithm, there are seven submatrices-calculations ($M1$ to $M7$) that are independent works that can be executed by parallel section constructs. The implementation in [31] only presented the performance of Strassen's algorithm in terms of $GFLOPS$ and speed-up computation excluding the measurement of energy/power consumption. The most suitable recursive stop-point was proposed in [31], and its tiling pattern was different from our proposed 2D blocking method, as shown in Fig. 3. Moreover, the OpenMP compiler directives and work-sharing constructs can be applied to improve task parallelism at the programming level.

A flowchart and the proposed implementation for power-efficient Strassen's algorithm are shown in Fig. 6 and Fig. 7, respectively. Intel AVX intrinsics have utilized with the OpenMP parallel *sections* constructs to increase the data and task parallelism on shared memory architecture. If the matrix size is equal to the defined recursive level, then the ***AVX_2D_Blocking_OMP()*** function is executed as proposed in Fig. 4. Otherwise, the recursive procedure continues to subdivide both the input submatrices A and B, and a new memory allocation has required for each level of recursion. The computation to obtain $m1$, $m2$, $m3$, $m4$, $m5$, $m6$, $m7$, $c00$, $c01$, $c02$, and $c03$ has been performed by independent tasks via the ***#pragma omp parallel sections nowait*** directive. After the results of each level of computation are obtained, all of the allocated memory for temporal storage is deallocated by using alignment function; *set_aligned_free()*.

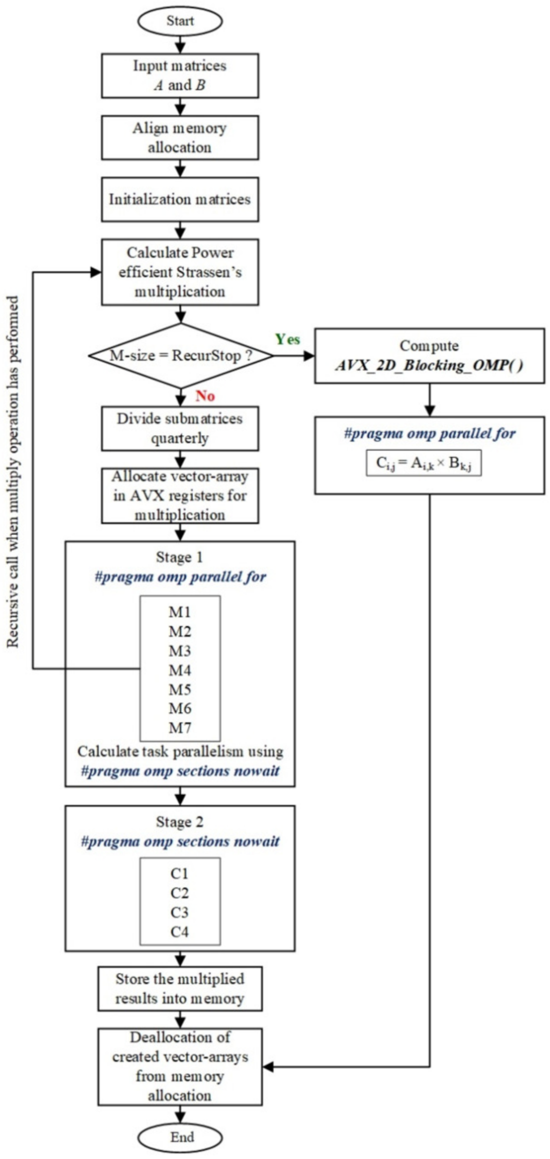
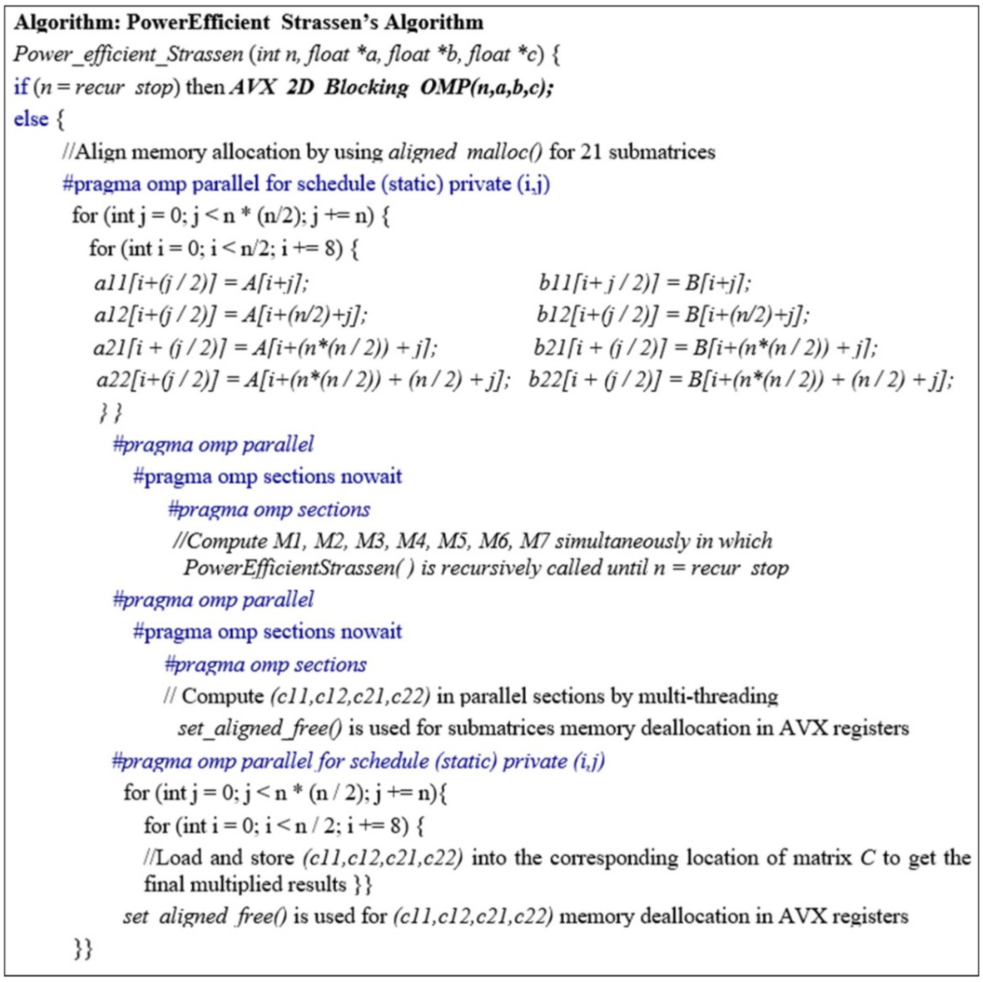

Start
Input matrices A and B
Align memory allocation
Initialization matrices
Calculate Power efficient Strassen's multiplication
M-size = RecurStop ?
Yes
No
Compute AVX_2D_Blocking_OMP( )
#pragma omp parallel for
$C_{i,j} = A_{i,k} \times B_{k,j}$
Divide submatrices quarterly
Allocate vector-array in AVX registers for multiplication
Recursive call when multiply operation has performed
Stage 1
#pragma omp parallel for
M1
M2
M3
M4
M5
M6
M7
Calculate task parallelism using #pragma omp sections nowait
Stage 2
#pragma omp sections nowait
C1
C2
C3
C4
Store the multiplied results into memory
Deallocation of created vector-arrays from memory allocation
End

Fig. 6: *Flowchart of 2D blocking in the multiplication kernel of power-efficient Strassen's algorithm.*

Firstly, all vector registers are set to be zero using $set\_zero\_matrix()$ before random data allocation is initialized with the size of defined matrices. Since most caches can store the vectors in a register with 32 or 64 bytes aligned vector arrays. Thus, when the vectors are aligned in the cache lines, the aligned data allocation is faster than the unaligned load/store operations since there does no need to be zero padding at the end of each row allocation. Data alignment increases the efficiency of data loads and stores to and from the processor [27]. In our implementation, the data from matrices A and B are allocated with the aligned value which must be an integer power of two. The float vector arrays for matrices A, B, and C with the pointers returned are not aliased which ensures the memory allocation has been aligned correctly by



```
Algorithm: PowerEfficient Strassen's Algorithm
Power_efficient_Strassen (int n, float *a, float *b, float *c) {
if (n = recur stop) then AVX 2D Blocking OMP(n,a,b,c);
else {
        //Align memory allocation by using aligned malloc() for 21 submatrices
        #pragma omp parallel for schedule (static) private (i,j)
         for (int j = 0; j < n * (n/2); j += n) {
            for (int i = 0; i < n/2; i += 8) {
             a11[i+(j / 2)] = A[i+j];                           b11[i+ j / 2)] = B[i+j];
             a12[i+(j / 2)] = A[i+(n/2)+j];                     b12[i+(j / 2)] = B[i+(n/2)+j];
             a21[i + (j / 2)] = A[i+(n*(n / 2)) + j];           b21[i + (j / 2)] = B[i+(n*(n / 2)) + j];
             a22[i+(j / 2)] = A[i+(n*(n / 2)) + (n / 2) + j];   b22[i + (j / 2)] = B[i+(n*(n / 2)) + (n / 2) + j];
             } }
              #pragma omp parallel
                #pragma omp sections nowait
                    #pragma omp sections
                    //Compute M1, M2, M3, M4, M5, M6, M7 simultaneously in which
                     PowerEfficientStrassen( ) is recursively called until n = recur stop
              #pragma omp parallel
                #pragma omp sections nowait
                    #pragma omp sections
                    // Compute (c11,c12,c21,c22) in parallel sections by multi-threading
                      set_aligned_free() is used for submatrices memory deallocation in AVX registers
              #pragma omp parallel for schedule (static) private (i,j)
               for (int j = 0; j < n * (n / 2); j += n){
                 for (int i = 0; i < n / 2; i += 8) {
                 //Load and store (c11,c12,c21,c22) into the corresponding location of matrix C to get the
                 final multiplied results }}
               set aligned free() is used for (c11,c12,c21,c22) memory deallocation in AVX registers
         }}
```

Fig.7: *Power-efficient Strassen's algorithm using 2D blocking with OpenMP.*

using the *_aligned_malloc* function. Our contributions are different from the previous [1][4][10][15][29] since we focused on high-performance energy efficiency ($GFPW$) on parallel Strassen's algorithm.

6. EXPERIMENTAL RESULTS

The proposed power-efficient algorithm has been tested on the Core *i9-7900X* machine with a *64-bit Windows 10* operating system. The test machine has *64* gigabytes of memory. The algorithm has been implemented using *Intel C-compiler 2019.* We implemented our algorithm and tested it versus the matrix-matrix multiplication function provided by the *Eigen* (version 3.3.7) [32] and the *OpenBLAS* (version 0.3.13) [33] which are high-level C++ libraries of template headers for linear algebra, matrix, and vector operations. Every configuration has compiled using the $O2$ optimization option. Each test configuration has evaluated using ten different square matrix sizes ranging from *1024× 1024* to *10240× 10240*. To configure the *Eigen* library [32] for multiplication kernel, which is a high-level C++ library of template headers for linear algebra, matrix, and vector operations. The two header files of sparse and dense matrix multiplication can be declared as follows:

$\#include < Eigen/Sparse >$
$\#include < Eigen/Dense >$

Since the proposed matrix-matrix multiplication focuses on dense matrix multiplication, $\#include < Eigen/Dense >$ must be declared when it was configured for multiplying random values of two matrices (A and B). The Matrix class inherits a base class, MatrixBase, and the initial declarations are required as follows:

$Eigen :: MatrixXf a;$
$Eigen :: MatrixXf b;$

To initialize random values ($n \times n$) for each matrix, since Eigen has a built-in constructor called, $MatrixXf :: Random(n, n)$, which is $Matrix :: Matrix(int)$, in $src/Core/Matrix.h$. It supports multi-threading since Eigen will launch as many threads as cores reported by OpenMP's function call.

Strassen's algorithm is faster than the standard matrix multiplication algorithm since the number of multiplication operations is reduced instead of **eight** to **seven** multiplications when 4×4 block-wise matrix multiplication is executed [4]. It is based on the divide and conquer mechanism to subdivide the matrices A and B into four sub-matrices of size $N/2\times N/2$ recursively and then calculate the seven equations ($M1-M7$) to get the values of C_{ij}. Typically, we still need **eight** multiplications of matrix blocks to calculate the C_{ij} matrices with $O(n^3)$, the asymptotic complexity for multiplying matrices of size $N=2^n$ using Strassen's algorithm is $O(n^{2.8074})$ [4].

The original Strassen's algorithm uses the seven equations ($M1-M7$) by calculating scalar matrix-matrix multiplication which does not include parallelism. Even though previous methods [30][31] used AVX intrinsic, titling and loop unrolling, the data accessing pattern of our 2D blocking matrix multiplication differs in the multiplication kernel of parallel Strassen's algorithm. Due to the optimized data accessing pattern from matrix A and B, the data transferring time can be reduced, which leads to increase performance and reduce energy consumption for parallel Strassen's algorithm.

In previous papers [5][6][10][27][31], they did not test their algorithm with the intensive examination of the energy/power consumption of Strassen's algorithm. In the proposed paper, the total number of additions/subtractions required for Strassen's algorithm at each recursive level can be calculated by using the proposed mathematical equations (Eq. 2, 3, 4, 5) which are presented in section 5.

We first measured the *GFLOPS* performance by comparing other libraries, and the results are shown in Fig. 8. Our algorithm is faster than *Eigen* and *OpenBLAS* in all configurations. It is, on average **4.5** and **4.1** times faster than *Eigen* and *OpenBLAS*, respectively.

Moreover, the energy and power consumption of both algorithms are measured using the *Intel power gadget* tool, a software-based power consumption monitoring tool designed for Intel's Core processors and is used for calculating real-time processor package power information using the energy counters in the processor [26]. In each configuration, the energy and power consumptions were collected ten times, and the average results are shown in Fig. 9 and Fig. 10. Our proposed algorithm consumes less energy than both *Eigen* and *OpenBLAS* libraries in every configuration. It consumes on average, **4.25** and **3.67** times less energy than *Eigen* and *OpenBLAS* libraries, respectively, as shown in Fig. 9. However, when power consumption was measured, our algorithm surpassed both *Eigen* and *OpenBLAS* only when the matrix sizes were *10240* × *10240*, as shown in Fig. 10. Its power consumption is on average **1.19** and **1.18** times higher than *Eigen* and *OpenBLAS*, respectively.

Since power consumption is the amount of energy used per unit of time, our algorithm is faster but it consumes more power than both *Eigen* and *OpenBLAS* because of its higher degree of parallelism. For this case, we utilized another way to measure these two algorithms based on their power consumption and with the awareness of their processing speed as well. Therefore, we used the *GFLOPS per Watt (GFPW)* to measure the *GFLOPS* performance of the proposed algorithm versus its power consumption.

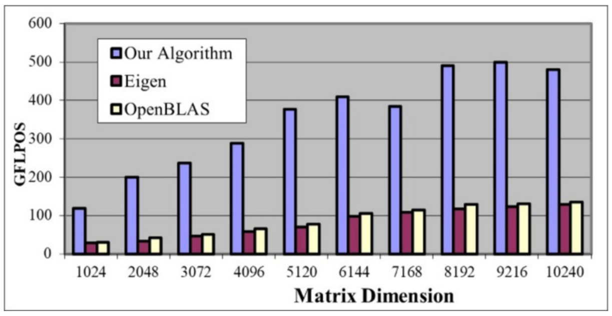


Fig.8: *GFLOPS performance of the proposed algorithm versus Eigen and OpenBLAS.*

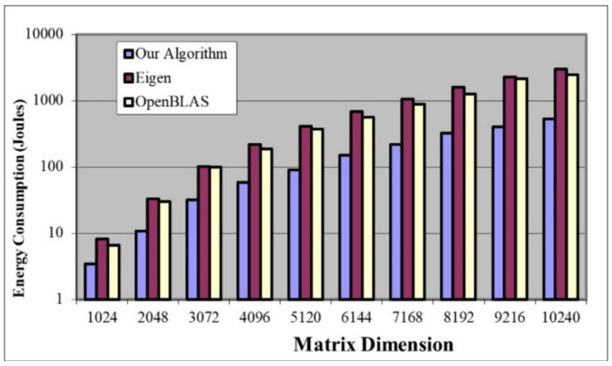


Fig.9: *Comparison of average energy consumption of proposed algorithm versus Eigen and OpenBLAS.*

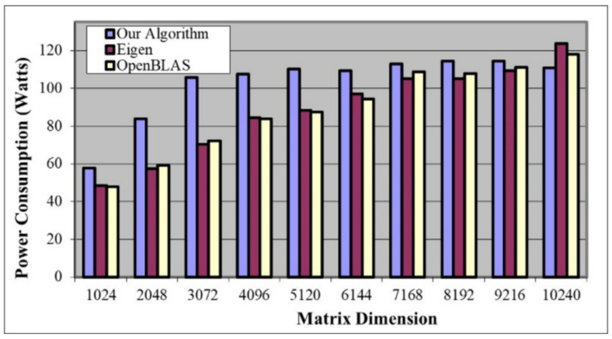


Fig.10: *Comparison of average power consumption of proposed algorithm versus Eigen and OpenBLAS.*

Table 2 shows the measurement methods that we utilized in our experiments. We further analyzed the *last level cache (LLC)* miss count of our proposed algorithm with the *Intel VTune Profiler* tools [34]. In this test, the automatic recursive level selection

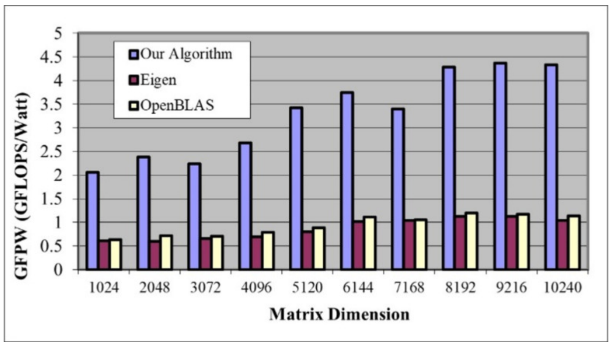


Fig.11: *GFPW performance of our proposed algorithm versus Eigen and OpenBLAS.*

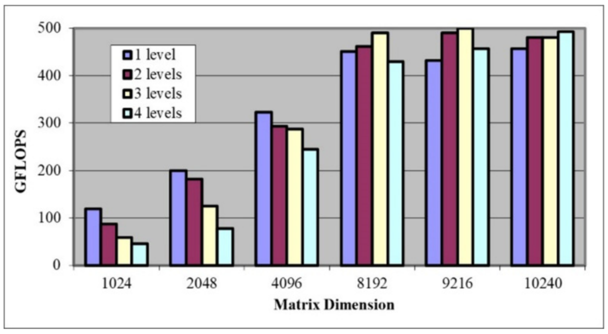


Fig.12: *GFLOPS performance of the proposed algorithm at different levels of recursion.*

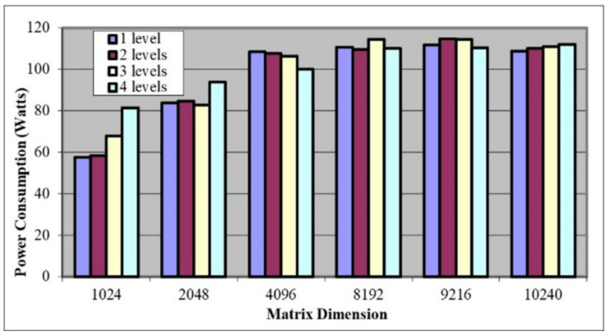


Fig.13: *Power consumption of our proposed algorithm at a different level of recursion.*

has turned off, and four different recursive levels were manually defined for the algorithm. The *GFLOPS* performance, the power consumption, and the *LLC* miss counts of each configuration were evaluated, and the results are shown in Fig. 12, Fig. 13, and Table 3.

While the power consumption of each test was measured, its *GFLOPS* performance was also collected at the same time, and its *GFPW* was calculated accordingly. Fig. 11 shows the *GFPW* performance of both algorithms. Although the power consumption of our proposed algorithm is higher than those of both *Eigen* and *OpenBLAS* when the matrix sizes are smaller than *10240×10240*, the GFPW performance of our algorithm is higher than that of those of *Eigen* and *OpenBLAS* in all configurations. The *GFPW* performance of our proposed algorithm is on average **3.78** and **3.47** times higher than those of *Eigen* and *OpenBLAS*, respectively.

Table 2: *Types of measurement methods utilized in our experiments.*

Type of Measurement	Formula	Explanations
Processing performance	$GFLOPS = \frac{2N^3}{t \times 10^9}$	$GFLOPS$ = Giga Floating-point Operation per Second N = the dimension of the matrix of size $N \times N$ t = Execution time in seconds 10^9 = the units of Giga (G)
Performance and power consumption ratio	$GFPW = \frac{GFLOPS}{Power}$	$GFPW = GFLOPS$ per watt $Power$ = Power consumption ($Watt$)

Table 3: *LLC miss counts of our implementation of Strassen's algorithm.*

Matrix Size	Number of recursive levels			
	1 level	2 levels	3 levels	4 levels
1024×1024	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2048×2048	0.00E+00	0.00E+00	0.00E+00	0.00E+00
4096×4096	0.00E+00	0.00E+00	0.00E+00	0.00E+00
8192×8192	1.62E+09	1.24E+09	1.22E+09	1.57E+09

Table 3 shows that when the matrix size is small, there is no *LLC* miss for all levels of recursion. However, the *LLC* miss occurs when the matrix size becomes larger, as shown in the last row (*8192×8192*) of Table 3. Since when the matrix size is small, all the data can be kept inside the cache, and the CPU core can access them as fast as possible. Therefore, the increase of recursive level increases the overhead of memory allocation/deallocation, resulting in less *GFLOPS* performance, as shown in Fig. 12. This evidence helps us better understand why the *GFLOPS* performance decreases with the increased level of recursion, especially when the matrix dimension is *1024*, *2048*, and *4096*.

When the matrix size is *8192×8192*, the *LLC* miss occurs and decreases with the increase of recursive levels. It reaches the minimum value when the recursive level is **three** but returns to increase when the recursive level is **four** or higher, as shown in Table 3. The reason is because it is impossible to keep all the data in the cache at the same time when the matrix size is large. Therefore, each level of recursion in Strassen's algorithm divides the source matrices into smaller sub-matrices, resulting in a gradual decrease in the *LLC* miss as the level of recursion increases. After the recursive level is four or higher, memory allocation/deallocation overheads begin to have a greater impact, leading to a decrease in *GFLOPS* performance.

7. DISCUSSION

The experimental results reveal that when 2D blocking is applied along with AVX512 intrinsics, OpenMP, and loop-unrolling can save energy and power consumption of Strassen's algorithm among different processing cores. We have compared the energy and power consumption of our implementation of Strassen's algorithm with the matrix-matrix multiplication function provided by *Eigen* and *OpenBLAS* libraries. In terms of GFLOPS performance, our algorithm obtains better computational speed than *Eigen* and *OpenBLAS* libraries in every configuration. It is on average **4.5** and **4.1** times faster than *Eigen* and *OpenBLAS*, respectively. It consumes on average, **4.25** and **3.67** times lower energy than *Eigen* and *OpenBLAS*, respectively, however, its power consumption is on average **1.19** and **1.18** times higher than that of *Eigen* and *OpenBLAS*, respectively.

Our high-speed 2D blocking matrix-matrix multiplication comes from four factors: **1)** the utilization of loop unrolling makes our algorithm can utilize all the AVX engines in each processing core as much as possible, **2)** the efficiency of arithmetic vectorization using AVX512 intrinsics allows more data to be processed at the same time using SIMD data processing instructions, **3)** the new optimized data allocation pattern allowing data in the cache to be reused as much as possible, therefore reducing main memory waiting time, and **4)** the utilization of OpenMP directives allows all the cores can process the data simultaneously, resulting in a high degree of parallelism.

The main issue of Strassen's algorithm is its overhead of extra memory allocation for sub-matrices at each recursive level, combined with the requirement of sub-matrices data transferring between each level of recursion. As a result, too many recursive calls can degrade the algorithm's performance.

In our proposed algorithm, we applied AVX512 registers with a new data allocation pattern for matrix B to reduce the number of data accessed from memory at each loop. Once the data are loaded from matrix B into the cache, it does not need to reload them when multiplying with the new items of a_1 and a_2. As a result, not only the computation speed is increased but also the power consumption of memory reading is reduced as well.

The reason that our algorithm consumes lower energy than the Eigen and OpenBLAS comes from three factors: **(i)** the utilization of AVX512 reduces the number of memory read operations by **8** times. Eight double-precision floating-point data can be read from main memory using only a single vector read operation, while each conventional scalar read procedure can read 64-bit double-precision floating-point data at a time, resulting in less energy consumption, **(ii)** an optimal number of recursive calls reduces the burden of unnecessary memory allocation/deallocation and data transferring for each level of recursion, and **(iii)** the proposed data allocation pattern with two types of loop unrolling factors (such as UF_a and UF_b) allowing data in the cache to be reused as much as possible and a new 2D blocking for multiplication kernel, therefore reducing the amount of main memory access which consumes more energy.

8. CONCLUSION

We have proposed an effective implementation for matrix-matrix multiplication based on Strassen's algorithm. Experimental results reveal that our algorithm consumes, on average, **4.25** and **3.67** times lower energy than the multiplication function provided by *Eigen* and *OpenBLAS* libraries, respectively. The *GFPW* performance of our proposed algorithm is on average **3.78** and **3.47** times higher than that of *Eigen* and *OpenBLAS* libraries, respectively.

Our main contributions include:

- The new 2D blocking pattern is proposed to enable the data to be reused in the cache as much as possible, as a result, the number of slow memory access is considerably decreased.

- The selection of an optimal number of recursive calls reduces the burden of unnecessary memory reallocation/deallocation and data transferring for each level of recursion and it provides the best sub-block size of the 2D blocking data allocation.

- The utilization of AVX intrinsics not only reduces the number of instructions to perform on the data but also reduces the number of memory loads/stores. In addition, the processing time is reduced due to its native parallel computation.

- The effect of loop-unrolling increases the utilization of all the execution units inside the CPU.

- The performance matrix called *GFPW* is utilized to measure the computational performance with the awareness of power consumption.

From this research, we conclude that the performance and energy consumption of parallel computing is mainly depend not only on the number of load/stores operations when accessing from main memory but also on the optimized parallelism of an algorithm. Therefore, a proposed optimized 2D blocking multiplication is very useful for all the energy-aware matrix-matrix calculations on multicore architecture.

ACKNOWLEDGEMENTS

This work was supported by the Higher Education Research Promotion and the Thailand's Education Hub for Southern Region of ASEAN Countries Project Office of the Higher Education Commission. (TEH-AC 048/2016)

References

[1] K. Livingston, A. Landwehr, J. Monsalve, S.Zukerman ,B. Meister and G.R. Gao, "Energy Avoiding Matrix Multiply," *International Workshop on Languages and Compilers for Parallel Computing, LCPC 2016: Languages and Compilers for Parallel Computing*, pp 55-70, vol. 10136, Springer, 2017.

[2] Y. Qin, G. Zeng, R. Kurachi, Y. Li, Y. Matsubara and H. Takada, "Energy-Efficient Intra-Task DVFS Scheduling Using Linear Programming Formulation," in *IEEE Access*, vol. 7, pp. 30536-30547, 2019.

[3] T. Jakobs and G. Rünger, "On the Energy Consumption of Load/Store AVX Instructions," *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Poznan, pp. 319-327, 2018.

[4] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Strassen's Algorithm Reloaded," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Article 59, pp. 690–701, Salt Lake City, UT, USA, 2016.

[5] J.Y. Huang, C.D. Yu, R.A. van de Geijn and A. Robert, "Strassen's Algorithm Reloaded on GPUs," *ACM Transactions on Mathematical Software*, vol. 46, no.1, pp.1-22, 2020.

[6] V. Arrigoni, F. Maggioli, A. Massini and E. Rodolà, "Efficiently Parallelizable Strassen-Based Multiplication of a Matrix by its Transpose," *50th International Conference on Parallel Processing: ICPP 2021*, ACM, Code 172277, Virtual (Online), 9-12 August 2021.

[7] Y. Tang and W. Gao, "Processor-Aware Cache-Oblivious Algorithms," *50th International Conference on Parallel Processing: ICPP 2021*, ACM, Code 172277, Virtual (Online), 9-12 August 2021.

[8] V.B.Kreyndelin and E.E.Grigorieva, "Fast matrix multiplication algorithm for a bank of digital filters," *2021 Systems of Signal Synchronization, Generating and Processing in Telecommunications, SYNCHROINFO 2021*, Article number 9488350, Svetlogorsk, Kaliningrad Region, June-July 2021.

[9] H. Kang, H.C. Kwon, and D. Kim, "HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs," *Computing*, vol. 102, pp. 2607-2631, 2020.

[10] A. J. Kawu, A. Yahaya Umar and S. I. Bala, "Performance of one-level recursion parallel Strassen's algorithm on dual core processor," *2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON)*, Owerri, pp. 587-591, 2017.

[11] L. Chen, P. Wu, Z. Chen, R. Ge, and Z. Zong, "Energy Efficient Parallel Matrix-Matrix Multiplication for DVFS-enabled Clusters," *2012 41st International Conference on Parallel Processing Workshops*, Pittsburgh, PA, pp. 239-245, 2012.

[12] H. Zamani, Y. Liu, D. Tripathy, B. Laxmi, and C. Zizhong, "GreenMM: energy efficient GPU matrix multiplication through undervolting," in *Proceedings of the ACM International Conference on Supercomputing*, New York, USA, pp. 308–318, 2019.

[13] K. Kashif Nizam, H. Mikael, N. Tapio, Jukka K. Nurminen, and Zhonghong Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no.2, page1-26, April 2018.

[14] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-Management architecture of the intel microarchitecture Code-Named sandy bridge," in *IEEE Micro*, vol. 32, no. 2, pp. 20–27, March-April 2012.

[15] T. Jakobs, M. Hofmann, and G. Runger, "Reducing the power consumption of matrix multiplications by vectorization," in *Proceedings of the 19th IEEE International Conference on Computational Science and Engineering (CSE 2016)*, pp. 1–8, August 2016.

[16] J. Haj-Yahya, A. Mendelson, Y. Ben-asher and A. Chattopadhyay, "Compiler-Directed Energy Efficiency," *Springer*, pp. 107-133, 2018.

[17] K. Momcilo, K. Jelena, P. Miroslav and K. Vlado, "Instructions energy consumption on a heterogeneous multicore platform," *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems ECBS'17*, Article No.: 10, pages 1–10, August 2017.

[18] H. David Money, H. Sarah L, "Digital Design and Computer Architecture," *Elsevier*, 2007.

[19] H. David, C. Fallin, E. Gorbatov, U. Hanebutte and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," *Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011 and Co-located Workshops*, pp. 31-40, 2011.

[20] M. Arjmandi, N. Mo. Balouchzahi, K. Raahemifar, M. Fathy and A. Akbari, "Reduction of Power Consumption in Cloud Data Centers via Dynamic Migration of Virtual Machines," *International Journal of Information and Education Technology*, vol. 6, no.4, pp. 286-290, 2016.

[21] M. Bharathwaj, S. Dharun, B. Akilesh, Mrs. M. Hema, "Reduction of Power Consumption Using Virtual Machine Consolidation in Cloud," *International Journal of Scientific Research and Review*, vol. 7, no. 3, pp. 246-249, 2018.

[22] Z. Zhou, Zg.Hu, Jy. Yu, et al. "Energy-efficient virtual machine consolidation algorithm in cloud data centers," Journal of Central South Univer-

sity, vol. 24, pp. 2331–2341, 2017.

[23] V.A Korthikanti and G.Agha, "Energy-Performance Trade-off Analysis of Parallel Algorithms," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 3, pp. 167-176, September 2011.

[24] J. Thiyagalingam, "Alternative array storage layouts for regular scientific programs," *Ph.D. Thesis*, 2005, `https://www.doc.ic.ac.uk/~phjk/PhDThesesLocalCopies/JeyanPhD.pdf`.

[25] Intel Corporation, "Intel® Architecture Instruction Set Extensions and Future Features," [Online]. Available: `https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programmingreference.pdf`.

[26] Intel Corporation, "Intel® Power Gadget," [Online]. Available: `https://software.intel.com/content/www/us/en/develop/articles/intel-power-gadget.html`.

[27] A. J. Kawu, A. Yahaya Umar and S. I. Bala, "Performance of one-level recursion parallel Strassen's algorithm on dual core processor," *2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON)*, Owerri, pp. 587-591, 2017.

[28] Barbara M. Chapman, Gabriele Jost, Ruud van der Pas, "Using OpenMP: Portable Shared Memory Parallel Programming," pp. 57, United States of America, MIT Press, 2008.

[29] S.Maleki, Y.Gao, M.J.Garzarń, T.Wong and D.A. Padua, "An Evaluation of Vectorizing Compilers," *2011 International Conference on Parallel Architectures and Compilation Techniques*, Galveston, TX, pp. 372-382, 2011.

[30] N. Z. Oo and P. Chaikan, "The Effect of Loop Unrolling in Energy Efficient Strassen's Algorithm on Shared Memory Architecture," *2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1-4, 2021.

[31] N. Z. Oo and P. Chaikan, "An effective implementation of Strassen's algorithm using AVX intrinsics for a multicore architecture," *Songklanakarin Journal of Science & Technology*, vol. 42, no. 6, pp. 1368-1376, 2020.

[32] Eigen [Online]. Available: `https://eigen.tuxfamily.org/Eigenindex.php?title=Main_Page`

[33] [OpenBLAS [Online]. Available: `https://osdn.net/projects/sfnet_openblas/downloads/v0.3.13/OpenBLAS\%200.3.13\%20version.tar.gz/`

[34] Intel Corporation. "Intel® VTune$^{\text{TM}}$ Profiler Performance Analysis Cookbook," [Online]. Available: `https://www.intel.com/content/www/us/en/develop/documentation/vtunecookbook/top.html`.

Nwe Zin Oo was born in Myanmar in 1987. She received her B.C.Sc., B.C.Sc.(Hons:), and M.C.Sc., in computer science from the University of Computer Studies, Yangon, in 2006, 2007, and 2010 respectively. Currently, she obtains TEH-AC scholarship for her Ph.D. degree in the Department of Computer Engineering from Prince of Songkla University, Thailand. Her position in Myanmar is a lecturer in the Faculty of Information Technology Supporting and Maintenance, University of Computer Studies, Myeik. Her research interests include natural language processing, parallel programming, high-performance computing, and controls and automation.

Panyayot Chaikan was born in Thailand in 1976. He received his B.Eng. and M.Eng, in computer engineering and electrical engineering from the faculty of Engineering, King Mongkhut's Institute of Technology Ladkrabang, Thailand, in 1999 and 2002 respectively. He obtained his Ph.D. degree in computer engineering in 2010 from Prince of Songkla University, Thailand. Currently, he has an associate professor position in the Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University, Thailand. His research interests include parallel processing, embedded systems, image processing, and pattern recognition.