A Computationally Efficient Algorithm for the 2D Covariance Method

Oded Green*
Georgia Institute of Technology
College of Computing
Atlanta, Georgia, USA
ogreen@gatech.edu

Yitzhak Birk
Technion - Israel Institute of Technology
Electrical Engineering
Haifa, Israel
birk@ee.technion.ac.il

ABSTRACT

The estimated covariance matrix is a building block for many algorithms, including signal and image processing. The Covariance Method is an estimator for the covariance matrix, favored both as an estimator and in view of the convenient properties of the matrix that it produces. However, the considerable computational requirements limit its use. We present a novel computation algorithm for the covariance method, which dramatically reduces the computational complexity (both ALU operations and memory access) relative to previous algorithms. It has a small memory footprint, is highly parallelizable and requires no synchronization among compute threads. On a 40-core X86 system, we achieve 1200X speedup relative to a straightforward single-core implementation; even on a single core, 35X speedup is achieved.

Keywords

Parallel algorithms, Covariance Method, Estimation, Inclusion-Exclusion Principle

1. INTRODUCTION

1.1 Background

The use of estimated covariance matrices originated in speech processing [16]. The Covariance Method is one method (estimator) for producing this matrix. Other applications requiring an estimated covariance matrix, not necessarily the one created by the Covariance Method, include nuclear magnetic resonance spectroscopy [3] and watermarking security based on a cryptanalysis perspective [5]. The Covariance Method's main rationale is minimizing the error in the estimate of a covariance matrix of a time series. Good estimates can give insights pertaining to the data periodicity, and enable fast and accurate analysis of the input data. The Covariance Method is a biased estimator, and its output is a Hermitian, positive semi-definite matrix. Unlike other methods, it is guaranteed to be non-singular. This often causes the Covariance Method to be preferred [17, 12].

The 2D Covariance Method, to be described later, is widely used for image processing because of the 2D dimensionality. It has also been applied elsewhere: signal processing applications wherein the signal's temporal correlation is needed [7], synthetic aperture radar range-azimuth focusing [15], smoothing spatial clutter by averaging over given transposed data [8], to perform 2D spectral analysis [11], and more. In all the aforementioned cases, the generation of the estimated covariance matrix using the Covariance Method is an important computational building block.

Efficient implementation of the 1D Covariance Method is trivial due to the trivially exploitable overlap of computations that need not be repeated. The 2D Covariance Method is significantly more challenging due to the non-trivial overlap, and its straightforward implementation requires multiple dense vector-vector multiplications. (The output of our algorithm is the same as that of the straightforward implementation of the Covariance Method.) In this work we show how the 2D Covariance Method can be computed efficiently, eliminating most of the redundant floating point additions, and how this can be done in conjunction with the effective parallelization presented in [6], which also eliminated redundant multiplications. The result is a sequential algorithm that is 30X-40X faster than the straightforward implementation (problem-size dependent); in conjunction with the parallelization of [6], we obtain an extremely efficient parallel algorithm: fine-grain parallelism, good load balancing, minimal repetition of work and no need for synchronization or communication among the parallel threads. Implementation results demonstrate the properties.

The remainder of the paper is organized as follows. In the rest of this section we introduce the Covariance Method, discuss related works, present some deficiencies of the straightforward algorithm and state the parallelization challenges. In Section 2 we present a different approach for implementing the Covariance Method [6] which partitions the work into non-trivial units. This approach exposes unique characteristics of the Covariance Method, which we then utilize for reducing the total number of operations. Section 3 introduces our new algorithm, followed by a detailed complexity analysis. In Section 4, we compare the measured performance of the different algorithms. These results are close to the expected ones, despite the fact that the complexity analysis accounts for operations, ignoring issues such as the

*Part of this work was done while Oded Green was at the Technion

Field Name	Description	Storage requirement
N_r	# rows in input matrix	1
N_c	# columns in input matrix	1
S_r	# rows in sliding window	1
S_c	# in sliding window	1
A	Input matrix	$N_r \times N_c$
$W^{k,l}$	Sliding window positioned at (k,l)	$S_r \times S_c$
$V^{k,l}$	Column stack of window $W^{k,l}$	$S_r \cdot S_c \times 1$
C	Output - estimated covariance matrix	$S_r \cdot S_c \times S_r \cdot S_c$
$C^{k,l}$	Partial sum array for $W^{k,l}$	$S_r \cdot S_c \times S_r \cdot S_c$
F^H	Conjugated transpose of F	

Table 1: Taxonomy of the Covariance Method

memory system. Section 5 offers some concluding remarks.

1.2 The Covariance Method

The input is an $N_r \times N_c$ matrix, A. It may represent pixel values of a 2D image, some other 2D spatial representation thereof (e.g., [7] and [19]), or a block ("aperture") within an image, with the estimated covariance matrix computed separately for each such block. Table 1 presents the taxonomy of the Covariance Method. The computation of the estimated covariance matrix using the Covariance Method is based on the movement of a sliding window W of size $S_r \times S_c$ over the input matrix, starting from the top left corner and moving to the bottom right corner. The considered window positions are all those that are fully encased within the boundaries of the input matrix. In the context of SAR imaging, this window is sometimes referred to as a sub-aperture. Fig. 1 depicts two 3×3 sliding window positions. A window in a given position is denoted $W^{k,l}$ where (k,l) is the position of the window's top left corner. There are $(N_r - S_r + 1) \times (N_c - S_c + 1)$ legal window positions. In the remainder of the paper, we refer to the estimated covariance matrix produced by the Covariance Method as the *output matrix*, and use *window* also to refer to a window position. Each of these windows can be considered as a sample from the sample space, the input matrix, where the dimensionality of the data is the size of the window.

For each window, $W^{k,l}$ is converted into a column stack $V^{k,l}$. This is followed by a dense vector-vector outer multiplication of $V^{k,l}$ by its conjugate transpose $(V^{k,l})^H$. The result is a partial sum of pairwise element products, used for the construction of the output matrix, and is stored in $C^{k,l}$.

Definition 1. *The estimated covariance matrix produced by the Covariance Method is given by :*

$$C = \sum_{k=1}^{N_r - S_r} \sum_{l=1}^{N_c - S_c} C^{k,l} = \sum_{k=1}^{N_r - S_r} \sum_{l=1}^{N_c - S_c} \vec{V}^{k,l} \cdot (\vec{V}^{k,l})^H. \quad (1)$$

This expression can also be regarded as a serial and straightforward algorithm for computing C. Note that C is Hermitian, being the sum of Hermitian matrices, so only the upper triangle or the lower triangle needs to be computed. Note also that some algorithms normalize this equation by the number of windows while others do not; this is an implementation detail that does change the scheme or performance of the algorithm.

Remark. In an actual implementation, the values can be written directly to C (the output matrix) rather than being stored in temporary matrices in order to reduce the memory footprint. However, this comes at the expense of a sparse memory access pattern and may reduce the effectiveness of caching. We assume the use of temporary matrices for presentation purposes.

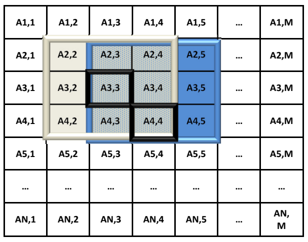


Figure 1: Two windows, silver (lighter) and blue (darker), and their overlapping sections. The black boxes around $A_{3,3}$ and $A_{4,4}$ refer to a product that needs to be computed for both the windows.

1.3 Related Work

In [7], the covariance matrix is used as part of the Minimum Variance Method (MVM) for Synthetic Aperture Radar (SAR) image processing. Due to the high complexity of computing the estimated covariance method for MVM, DeGraaf [7], suggested an alternative and computationally cheaper estimated covariance matrix as part of the Reduced-Rank MVM (RRMVM). However, this approach reduces the effectiveness of the algorithm by introducing more noise, making the Covariance Method preferable. The MVM algorithm uses an image of size $1.6k \times 1.6k$ for testing. The image is divided into $10,000$ 25×25 input matrices (there is overlap between the input matrices). This requires computing 10^4 estimated covariance matrices per image or per frame in an image sequence.

Yadin *et al.* [19] use images of size $8k \times 8k$ with the MVM algorithm, computing $25 \cdot 10^4$ estimated covariance matrices(25X more matrices than were required by [7]). Both papers show that MVM gives a better output image than using an FFT. Reducing the matrix computation time is important in view of the number of such computations. In fact, the amount of required computation may determine whether on-board computation (e.g., satellite) is practical, with interesting operational ramifications such as support of real-time decisions and adaptations.

In [11] the Covariance Method is compared with the Toeplitz-Block-Toeplitz method as part of the classic Capon estimator [4] and with APES (Amplitude and Phase) spectral estimator [13, 14]. The covariance method is experimentally shown to be more computationally demanding than these methods by about an order of magnitude. In [10], the Capon and APES are extended for the creation of a new estimator which again uses the Covariance Method. By providing a more computationally efficient implementation of the functionally-advantageous Covariance Method, we broaden its applicability, be it in terms of image size or frame rate.

Accelerators and HPC platforms such as the Cray XMT and NVIDIA's CUDA platform have been used for accelerating image processing algorithms. The Cray XMT2 was used for image segmentation based on a maxflow mincut approach on $32k \times 32k$ images [2]. CUDA is used for CT reconstruction [18]. Although we demonstrate our scheme on a multi-core X86 platform, the nature of the parallelizability (total independence) and the similarity of computation suggest that it can be beneficial on various acceleration platforms, including SIMD-oriented ones.

In [6], parallelization of the Covariance Method is studied.

An effective parallelization scheme is presented, and redundant multiplications are avoided. For facility of exposition, we will return to this later.

1.4 Deficiencies of the Straightforward Algorithm

The multiplication of $V^{k,l} \cdot (V^{k,l})^H$ in (1) is actually a multiplication of every element in each vector by every element in the other one. Therefore, when considering two sliding window positions $W^{k,l}$ and $W^{k',l'}$ such that both contain the product of some two elements x and y, both result matrices $C^{k,l}$ and $C^{k',l'}$ will contain the following products: $x \cdot \bar{x}, x \cdot \bar{y}, y \cdot \bar{x}$ and $y \cdot \bar{y}$. These products will also be added to the different partial sum matrices at different indices. Moreover, there is an overlap in the summation process as several products may be added to a specific index.

In Fig. 1, for example, the product $A_{3,3} \cdot \overline{A_{4,4}}$ is required for all windows containing these two elements. Two of these windows are depicted in Fig. 1. Note the different positions of these products relative to the upper left corner of the sliding window in $C^{k,l}$ and $C^{k',l'}$. This difference determines the indices to which the product contributes. The straightforward algorithm computes each product multiple times, once per target index. Now consider the products $A_{3,3} \cdot \overline{A_{3,3}}$ and $A_{4,4} \cdot \overline{A_{4,4}}$. Each of these products can be found in up to 9 different windows. Also, the sum of these products will be accumulated to the same indices in the output matrix more than once. The result is repetition of multiplications and additions. The challenge is to reduce the required number of operations without consuming much memory for temporary results or requiring inter-task synchronization that could jeopardize effective parallelization. Given that the different products impact different indices in the output matrix, the latter poses an additional challenge, namely to find the contribution pattern for a single product.

In Fig. 2, the dashed curve shows the number of elements in the estimated covariance matrix, and the solid curve shows the required number of floating point operations (multiplications as well as additions) as a function of a square-window size. Note that the number of floating point operations is considerably high due to redundant operations (both multiplications and additions), and the curve for the number of operations vs. window size has a bell like shape. The bell shape stems from the fact that the number of operations is the product of the operations per window positions and the number of such positions. When the window is very small, the former is very small; when it is very large, the latter is very small; the maximum is achieved with intermediate windows sizes.

The number of multiplications required by the straightforward algorithm is:

$$(N_r - S_r + 1) \cdot (N_c - S_c + 1) \cdot S_r^2 \cdot S_c^2. \qquad (2)$$

This is also the number of additions required by the straightforward algorithm, because each multiplication is carried out in the course of computing a sum of products.

1.5 Parallelization Challenges

Effective parallelization of the serial algorithm presents several challenges: 1) obviating the need for synchronization and atomic instructions, 2) limiting the required amount of memory for intermediate results, 3) utilizing all cores all the

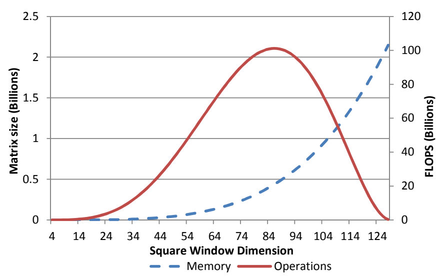


Figure 2: The dashed curve denotes the number of elements in the estimated covariance matrix. The solid curve denotes the number of floating operations need to compute the estimated covariance method for a given problem size using the straightforward formulation. The x-axis denotes the window length=width.

time (efficient load balancing), and 4) avoiding redundant computations. These must all be addressed concurrently.

Various intuitive parallelization approaches fail to meet all challenges. For example, assigning a different row of the output matrix to each core would result in redundant operations. An additional approach might be to give each of the P cores a near-equal number of windows to compute. In this approach, each core maintains a temporary output array for the accumulation process. When all the cores have completed, the temporary output arrays are summed up. This approach increases the memory requirements by a factor of P, which limits the problem size which the algorithm can be applied, be it directly because of memory size or due to very poor memory access time in view of insufficient cache size and very slow execution as a result. Furthermore, this approach does not avoid redundant computations. A different approach would be to assign any given product to a single task that would add the product to the multiple temporary output matrices. This approach requires synchronization and possibly atomic operations, which also reduces the scalability of the algorithm and can limit portability.

For algorithms such as MVM [7], where a large number of estimated covariance matrices need to be computed, a coarse grain approach can be taken whereby each core is responsible for computing the estimated covariance matrix for a different input. This, however, increases the memory footprint, and does not address the problem of redundant operations in the computation of any given matrix.

2. FINE-GRAIN MULTIPLICATION-EFFICIENT PARALLELIZATION

The key contribution of [6] was the discovery and formulation of interesting relationships between relative positions of input-matrix elements, dubbed *combinations*, pairwise products, and positions (indices) in the output matrix. Based on these, [6] went on to propose a partitioning of the output-matrix indices amongst cores such that any given index is constructed by a single core, thereby obviating the need for inter-core synchronization; yet, each core is assigned the union of the indices to which any of the products that it computes contributes, obviating the need to repeat any multiplication by multiple cores or to require cores to synchronize or communicate with one another. Although this algorithm significantly reduces the number of multiplications, however, it does not reduce the required number of additions. This algorithm offers the speedup resulting from parallelism, enabling a reduction in the latency of computing the output matrix. However, its contribution to computation throughput (when multiple independent matrices need to be computed) is limited, despite the savings in multiplications, because of the additions. The sequential speedup (relative to the straightforward implementation) on an X86 platform is in on the order of $2X-4X$, representing the savings in multiplications, an improved memory access pattern and a reduced memory footprint.

In the remainder of this section, we briefly describe this algorithm, which also serves as a starting point for our new algorithm. We include a summary of the key lemmas without any formal proofs. This section introduces relevant definitions that are also required by our new algorithm. The interested reader is referred to [6].

2.1 Product Based Partitioning

The challenge addressed in [6] [9] was partitioning the products among the cores such that any given output-matrix index is constructed by a single core, each product is computed only once and contributes to all the windows containing it, and cores need not communicate with one another or share data. The products are partitioned such that the relative position (row distance, column distance) of the elements of all products in any given group is the same, regardless of the window.

Definition 2. *Let A_{r_1,c_1} and A_{r_2,c_2} be two elements in the input matrix. Their inter-element distance (vector) is defined as:*

$$\Delta \triangleq (\Delta r, \Delta c) = (r_2 - r_1, c_2 - c_1). \tag{3}$$

Definition 3. *A* **combination** *is the set of all products of two input-matrix elements with the same inter-element distance; it is denoted by this distance, Δ, which must satisfy the following two conditions:*

$$-(S_r - 1) \leq \Delta r \leq S_r - 1 \wedge -(S_c - 1) \leq \Delta c \leq S_c - 1. \tag{4}$$

For example, the products of element pairs $(A_{3,3}, A_{4,4})$ and $(A_{5,5}, A_{6,6})$ both belong to combination $(\Delta r, \Delta c) = (1,1)$. Reversing the order of a pair, e.g., $(A_{4,4}, A_{3,3})$, gives a products that belongs to the combination $(\Delta r, \Delta c) = (-1,-1)$. The restrictions on the distances reflect the fact that we are only interested in products of elements that can be within the same $S_r \times S_c$ window.

Given that both products must be within the window, we define two disjoint sets of combinations. The first set comprises all the combinations wherein the first multiplicand is located at the top left corner of the window and the second element may be anywhere in the window. There are $S_r \cdot S_c$ possible positions of the second element. Each of these positions creates a different combination. The top left corner of the window is used for the purpose of finding the different combinations, whereas the position of the other multiplicand relative to the top left corner determines the indices in the output matrix to which this product will be added (accumulated). This first set of combinations, denoted POS, is

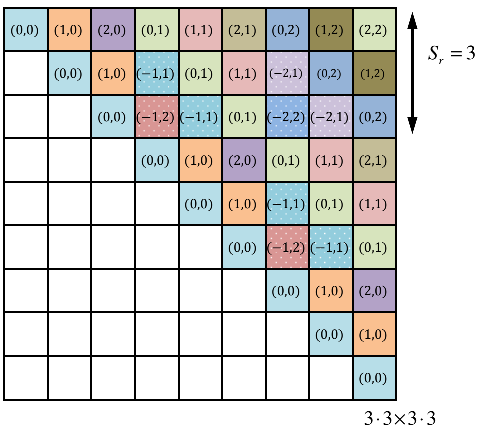


Figure 3: The following, 9×9, output matrix is for a 3×3 sliding window. The output matrix is partitioned into the different indices of the combinations. The combinations write to non overlapping sections of a specific diagonal. Each combination is depicted using a different color. All the combinations in POS have a solid colored background and the combinations of NEG have white dots in the background.

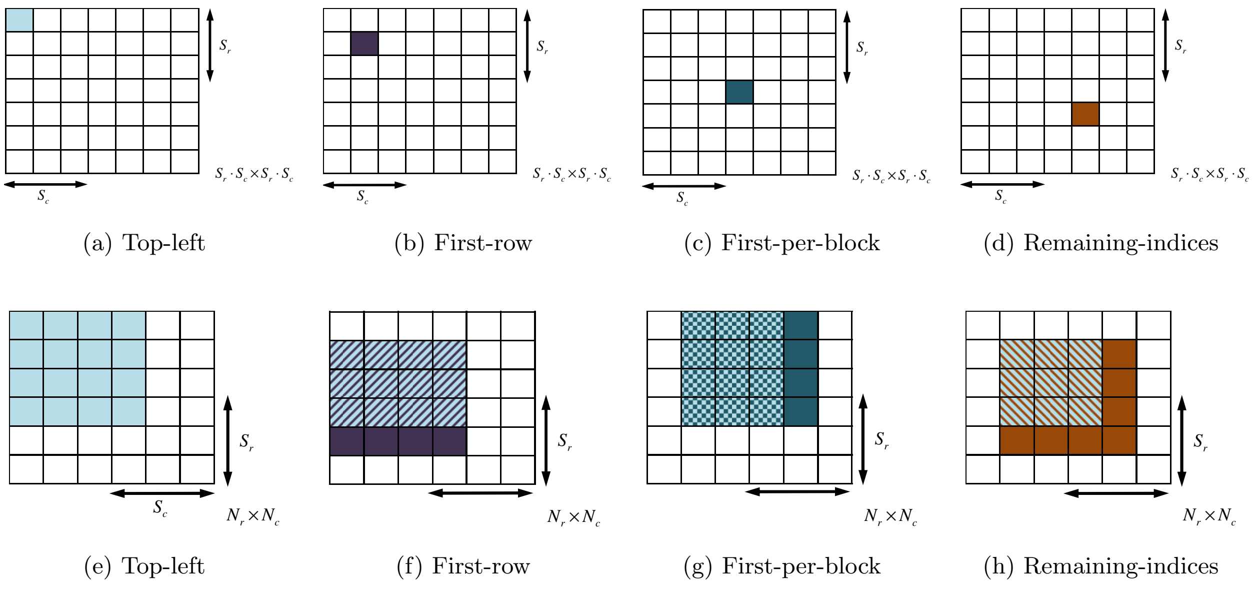
formally defined as:

$$POS = \{(\Delta r, \Delta c) | (0 \leq \Delta r \leq S_r - 1) \wedge (0 \leq \Delta c \leq S_c - 1)\}. \tag{5}$$

The second set comprises all combinations wherein the first element is in the first column and the second element is to the right and above the first element (rather than in the same row or below as in POS). There are $S_r - 1$ possible ways to place the first element. For each of these, the second element can be in $S_c - 1$ different places. This allows for a total of $(S_r - 1) \cdot (S_c - 1)$ different combinations. This second set of combinations, denoted NEG, is formally defined as:

$$NEG = \{(\Delta r, \Delta c) | (-(S_r - 1) \leq \Delta r \leq -1) \wedge (1 \leq \Delta c \leq S_c - 1)\}. \tag{6}$$

Let UC denote the set of unique combinations. Note that for all the combinations in UC, $\Delta c \geq 0$, whereas Δr can be negative, zero or positive. Based on the observation that the order of the element-pair affects the distance, an immediate question arises as to why the combinations in which $\Delta c < 0$ are not included in UC. The reason is that the estimated covariance matrix is Hermitian, so it suffices to compute the upper triangle or lower triangle sub-matrices. Any two inverse-distance combinations (e.g. $(\Delta r, \Delta c) = (a, b)$ and $(\Delta r, \Delta c) = (-a, -b)$) consist of the same element pairs. As the multiplication is done by taking the conjugate of one of the elements, we can utilize the trivial identity $A_{x',y'} \cdot \overline{A_{x,y}} = \overline{A_{x,y} \cdot \overline{A_{x',y'}}}$ to further reduce the number of actual multiplications. As can be seen, Δc is always positive, so the second element of any product is to the right of the first element; consequently, all results are written to the upper triangle. The products belonging to $\Delta c < 0$ combinations, which "own" the lower triangle, need not be computed.Accordingly, the set of unique combinations is specified by the joining of



(a) Top-left (b) First-row (c) First-per-block (d) Remaining-indices

(e) Top-left (f) First-row (g) First-per-block (h) Remaining-indices

Figure 4: Product centric approach for combination $(\Delta r, \Delta c) = (0, 0)$. These windows slightly vary (different height and width) for different combinations. (a)-(d) refer to the exact location in the output matrix. (e)-(h) refer to the set of products that are need to compute the indices in (a)-(d).

the two sets:

$$UC = POS \cup NEG. \tag{7}$$

The total number of combinations is:

$$|UC| = |POS| + |NEG| = S_r \cdot S_c + (S_r - 1) \cdot (S_c - 1). \tag{8}$$

In [9], several theorems are proved to support the following claims:
1) The combinations jointly cover the upper triangle of the output matrix.
2) Collision Freedom - the sets of write indices of two different combinations are disjoint.
3) An given combination writes to a specific set of indices that are all along the same diagonal. This is illustrated in Fig. 3.

Consequently the target indices of the combinations jointly cover the entire upper triangle of the output matrix, and any given index is written to by a single combination. Additionally, no synchronization is required among the cores. Furthermore, given that each combination is independent of the other combinations, these can be computed in parallel.

3. THE NEW ALGORITHM

The approach of [6] can be viewed as product centric. A product was computed exactly once, and was then added to all the output-matrix indices to which it contributes (i.e., belonging to its combination). This was done by sliding a window of the same dimensions as W around the product. Consequently, while the algorithm is parallelizable (by combination) and number of multiplications was held to the bare minimum, the number of additions was not reduced relative to the straightforward approach. In this section we present our new algorithm. We use the combination-based partitioning of [6], so parallelizability is retained. Our focus is on more efficient computation of the output matrix elements whose indices belong to a common combination. Specifically, our aim is to reduce the required number of additions. In the sequel, we consider a single combination and the computation of the output-matrix elements (indices) that belong to it.

Instead of considering each product and the indices to which it contributes, we consider an index (position) in the output matrix and all the products that contribute to its value. We show that there is a substantial overlap between the sets of products contributing to different same-combination indices, and that this overlap has a repetitive pattern that is somewhat similar to the Inclusion-Exclusion principle. Based on this, we devise a scheme for incrementally computing an index by starting from the value of the previously computed one and then adding and subtracting the non-overlapping products. In fact, we mostly do not need to add and subtract each non-overlapping product separately; instead, partial sums can be manipulated. We refer to this as a *product window* approach.

3.1 Combination $(\Delta r, \Delta c) = (0, 0)$

As an example, next consider the combination $(\Delta r, \Delta c) = (0, 0)$. This subsection gives the intuition behind the computation; this will subsequently be formalized, with Algorithm 1 stating the exact computation for $(\Delta r, \Delta c) = (0, 0)$ and more. The indices of $(\Delta r, \Delta c) = (0, 0)$ are the indices that make up the main diagonal, Fig. 3.

Consider the index $C_{1,1}$in the output matrix, this index is denoted in light blue in Fig. 4(a). Note that this is the top left index of the output matrix. Remember that this index is the sum of all the top left indices for all the temporary matrices $C^{k,l}$. Specifically, this index equals the sum of the products of $\overline{V^{k,l}(1,1)} \cdot \overline{V^{k,l}(1,1)}$ of all the windows. In Fig. 4(e), using light blue, we marked all the elements in the input matrix that are needed for computing $C_{1,1}$. Remember that each element is multiplied by its conjugate. For a different combination than (0,0) , Fig. 4(e) would be of a different dimension, as the number of window positions changes. Given that the sliding window must stay within the bounds of the input array, the different number of windows is $(N_r - (S_c - 1) \cdot (N_c - (S_c - 1)))$. Note that the matrices in the (a) and (e) do not have the same dimensions.

Now consider the multiplication of $V^{k,l}(2,1) \ \cdot \overline{V^{k,l}(2,1)}$. These products, for the different windows, also belong to combination (0,0). The sum of these products is written to $C_{2,2}$ as depicted in Fig. 4(b) marked in dark purple. As for the previous index $C_{1,1}$, all the needed products in the input array are marked in dark purple, Fig. 4(f). The products that are common to indices $C_{2,2}$ and $C_{1,1}$ are marked with blue and purple diagonal stripes.

The difference between the sums constituting these two indices is the sum of the first row from Fig. 4(e), which must be removed (subtracted), and the sum of the newly added row from Fig. 4(f), which must be added. This is repeated for all the product windows wherein the leftmost product is in the first column of the input matrix. There are $S_r - 2$ such windows, given that the first and top-left most window requires computing all the products. For all $1 \leq g \leq (S_r - 2)$, the following can be stated:

$$C_{g+1,g+1} = C_{g,g} + \sum row_{new} - \sum row_{old}. \qquad (9)$$

Note that $C_{g+1,g+1}$ depends on $C_{g,g}$ which depends on $C_{g-1,g-1}$. Therefore, it is preferable to compute in the order $C_{1,1}, C_{2,2}, C_{3,3}, ..., C_{S_r-1,S_r-1}$.

Algorithm 1 Parallel algorithm for computing the combinations in *POS*. The combinations in *NEG* are computed in a similar fashion.

```
$D_r \leftarrow N_r - S_r + 1$; $D_c \leftarrow N_c - S_c + 1$;
for $(\Delta r, \Delta C) \in POS$ in parallel do
    $ind_r \leftarrow +1$; $ind_c \leftarrow S_r \cdot \Delta c + \Delta r + 1$;
    // Top-left
    for $r = 1$ to $(N_r - S_r + 1)$ do
        for $c = 1$ to $(N_c - S_c + 1)$ do
            $C_{ind_r,ind_c} \leftarrow C_{ind_r,ind_c} + A_{i,j} \cdot \overline{A_{i+\Delta r,j+\Delta c}}$;
        end for
    end for
    // First-block
    $ind_r \leftarrow 2$; $ind_c \leftarrow S_r \cdot \Delta c + \Delta r + 2$; $sum \leftarrow 0$;
    for $r = 2$ to $E(\Delta r)$ do
        for $c = 1$ to $D_c$ do
            $sum \leftarrow sum + A_{D_r+r,c} \cdot \overline{A_{D_r+r+\Delta r,c+\Delta c}} - A_{r-1,c} \cdot \overline{A_{r-1+\Delta r,c+\Delta c}}$;
        end for
        $C_{ind_r,ind_c} \leftarrow C_{ind_r-1,ind_c-1} + sum$;
        $ind_r \leftarrow ind_r + 1$; $ind_c \leftarrow ind_c + 1$; $sum \leftarrow 0$;
    end for
    // First-Per-Block
    $ind_r \leftarrow S_r + 1$; $ind_c \leftarrow (S_r + 1) \cdot \Delta c + \Delta r + 1$; $sum \leftarrow 0$;
    for $c = 2$ to $B(\Delta c)$ do
        for $r = 1$ to $D_r$ do
            $sum \leftarrow sum + A_{r,DC+c} \cdot \overline{A_{r+\Delta r,DC+c+\Delta c}} - A_{r,c-1} \cdot \overline{A_{r+\Delta r,c-1+\Delta c}}$;
        end for
        $C_{ind_r,ind_c} \leftarrow C_{ind_r-S_r,ind_c-S_r} + sum$;
        $ind_r \leftarrow ind_r + S_r$; $ind_c \leftarrow ind_c + S_r$; $sum \leftarrow 0$;
    end for
    // Remaining-indices
    $ind_r \leftarrow S_r + 2$; $ind_c \leftarrow S_r \cdot \Delta c + S_r + \Delta r + 2$;
    for $r = 2$ to $E(\Delta r)$ do
        for $c = 2$ to $B(\Delta c)$ do
            $a \leftarrow A_{r-1,c-1} \cdot \overline{A_{r-1+\Delta r,c-1+\Delta c}}$
            $b \leftarrow A_{r-1,D_c+c} \cdot \overline{A_{r-1+\Delta r,D_c+c\Delta c}}$
            $c \leftarrow A_{D_r+r,h-1} \cdot \overline{A_{D_r+r+\Delta r,c-1+\Delta c}}$
            $d \leftarrow A_{D_r+r,D_c+c} \cdot \overline{A_{D_r+r+\Delta r,D_c+c\Delta c}}$
            $\Delta sumrow \leftarrow C_{ind_r-S_r,ind_c-S_r} - C_{ind_r-S_r-1,ind_c-S_r-1}$
            $C_{ind_r,ind_c} \leftarrow C_{ind_r-1,ind_c-1} + \Delta sumrow + a - b - c + d$;
            $ind_r \leftarrow ind_r + 1$; $ind_c \leftarrow ind_c + 1$;
        end for
        $ind_r \leftarrow r \cdot S_r + 2$; $ind_c \leftarrow (S_r) \cdot \Delta c + r \cdot S_r + \Delta r + 2$;
    end for
end for
```

Similarly, moving the product window to the right allows computing additional elements of the same combination. The first move of the product window computes C_{S_r+1,S_r+1} as depicted in Fig. 4(c). Each additional move of the product window to the right will compute elements of the following format: $C_{S_r \cdot h+1, S_r \cdot h+1}$ for $1 \leq h \leq S_c - 2$. These indices can be computed as follows:

$$C_{S_r \cdot h+1,S_r \cdot h+1} = C_{S_r \cdot (h-1)+1,S_r \cdot (h-1)+1} + \sum column_{new} - \sum column_{old}. \qquad (10)$$

Again, it is preferable to compute in the order $C_{1,1}$, $C_{S_r+1,S_r+1}, C_{2 \cdot S_r+1,2 \cdot S_r+1}, ..., C_{(S_c-1) \cdot S_r-1,(S_c-1) \cdot S_r-1}$.

Up to now, we have shown how to compute $S_r + S_c - 1$ indices of a specific combination. There still remain $S_r \cdot S_c - (S_r + S_c - 1)$ indices that need to be computed for the combination. These indices could be computed using the same techniques as discussed. However, there is a less computationally demanding approach to compute these indices, using additional overlapping information. In Fig. 4(h) we see the products common to the index in Fig. 4(d) and to $C_{1,1}$. Note that the overlap between the turquoise product-window in Fig. 4(g) and the brown product-window in Fig. 4(h) includes everything except for the top row and bottom row. In Fig. 5, the overlap of the same product window from Fig. 4(h) is shown with overlap of the additional product windows. The value of C_{S_r+2,S_r+2} includes products that are not part of the product window of C_{S_r+1,S_r+1}. Note that there is only a single product that was not computed as part of the other product windows. We show that using an inclusion-exclusion like principle we can compute this product window using previously computed values. We show this principle in steps. Given the overlap with the product window of C_{S_r+1,S_r+1}, we add this value to C_{S_r+2,S_r+2}. Obviously some corrections need to be made to this sum, as the first row (light blue) need not be considered and the last row (purple) needs to be considered. Note that the first row has one turquoise product in addition to the light blue products and the new row has one brown in addition to the purple products. We make the corrections in two phases. Similar to the process that was shown earlier, the sum of the new purple row minus the sum of the first light blue row is computed as follows:

$$\Delta sum_{row} = C_{2,2} - C_{1,1}. \qquad (11)$$

We handle entire row and column sums, so “corner” elements are handled twice. These must be handled individually. We next address this issue in detail. In the process of computing Δsum_{row}, it was assumed that the entire row overlaps with the turquoise product-window, when in fact the top-left element denoted as a in Fig. 5 is not part of the overlapping window. Given that it has been subtracted, it needs to be added back to the sum. The first product of the bottom purple row was also added to the sum. Therefore, it needs to be subtracted, this product is denoted c. a and c have corrected the summation process of (11), but two additional corrections remain to be considered. By using the product-window of C_{S_r+1,S_r+1}, an additional error was introduced into the summation process; this is the top-right turquoise product denoted as b in Fig. 5. This can be corrected by subtracting this product from the final sum. Finally, the brown product window has a new value that does not over-

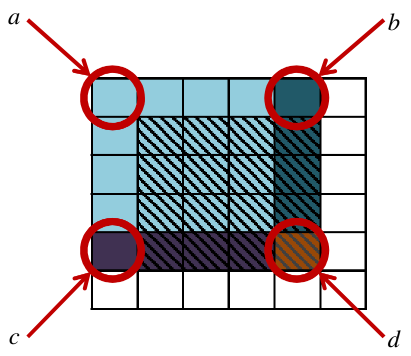


Figure 5: Zoom-in on the overlapping windows. a, b, c, and d denote unique elements that need to be added/subtracted individually with the new approach.

lap with any previous windows; this is denoted as d in Fig. 5. This value has to be added to the final summation. In summary, C_{S_r+2,S_r+2} can be written as follows:

$$C_{S_r+2,S_r+2} = C_{S_r+1,S_r+1} + \Delta sum_{row} + a - b - c + d \quad (12)$$

For C_{S_r+2,S_r+2}, a, b, c, and d are defined as follows:

$$a = A_{1,1} \cdot \overline{A_{1,1}}, \quad (13)$$

$$b = A_{1,N_c-S_c} \cdot \overline{A_{1,N_c-S_c}}, \quad (14)$$

$$c = A_{N_r-S_r,1} \cdot \overline{A_{N_r-S_r,1}}, \quad (15)$$

and

$$d = A_{N_r-S_r,N_c-S_c} \cdot \overline{A_{N_r-S_r,N_c-S_c}}. \quad (16)$$

This overlap procedure is the same for all remaining indices of the combination. Algorithm 1 provides the exact "recipe".

3.2 Remaining Combinations

In this subsection, we discuss how to compute the remaining combinations.

Definition 4. *The number of blocks in combination* $(\Delta r, \Delta c)$ *is* $B(\Delta c) = S_c - |\Delta c|$.

Definition 5. *The number of rows in a block in combination* $(\Delta r, \Delta c)$ *is* $E(\Delta r) = S_r - |\Delta r|$.

For the $(\Delta r, \Delta c) = (0, 0)$ example, $B(0) = S_c$ and $E(0) = S_r$.

Each combination can be divided into four unique groups:

Definition 6. *We denote the four different computation scenarios as Top-left, First-block, First-per-block, Remaining-indices.*

Pseudo-code for computing the combinations in POS can be found in Algorithm 1. Most of the explanations for computing these combinations are similar to the explanation given for combination $(\Delta r, \Delta c) = (0, 0)$. Table 2 presents the number of operations for each combination based on the foregoing definitions.

For the sake of brevity, we only provide necessary observations for computing the remaining combinations. The first group, Top-left, comprises a single index in the output matrix. The index of the Top-left depends on the combination and the window dimensions. Computation of this index appears in Line 3 of Algorithm 1. This single index can be more computationally demanding than computing all the remaining indices of a combination, because here we prepare the partial sums that are later used to incrementally modify the value of one index in order to obtain that of the next one. This will be discussed in depth in the next subsection. The Top-Left index dictates which diagonal in the output matrix will be affected by the combination. The Top-Left for combinations in NEG will be computed slightly differently, but it too designates the target diagonal.

Element type	*Multiplications*	*Additions*	*# elements of type*
Top-left	$(N_r - S_r + 1) \cdot (N_c - S_c + 1)$	$(N_r - S_r + 1) \cdot (N_c - S_c + 1)$	1
First-block	$2 \cdot (N_c - S_c + 1)$	$2 \cdot (N_c - S_c + 1)$	$E(r) - 1$
First-per-block	$2 \cdot (N_r - S_r + 1)$	$2 \cdot (N_r - S_r + 1)$	$B(c) - 1$
Remaining-indices	4	7	$(E(r) - 1) \cdot (B(c) - 1)$

Table 2: The number of operations needed for each combination based on the four types of product-window shift.

3.3 Complexity Analysis

In this section we analyze the work complexity of the new method. For simplicity, we analyze the complexity for the combinations in the set POS. The number of operations for a combination (a, b) is equal to the number of operations in combination $(a, -b)$. Also, $|POS| > |NEG|$, meaning that computing the combinations in POS is more work than computing those in NEG. Therefore, deriving the complexity of POS and doubling it yields a conservative approximation. We examine the four types of window shifts, and derive the complexity for each. Finally, we add up the complexities of the different window shifts and double it. For three out of the four types of shifts, the numbers of additions and multiplications are the same.

For the first type of indices, Top-Left, the number of multiplications over all the combinations is:

$$\sum_{\Delta r=0}^{S_r-1} \sum_{\Delta c=0}^{S_c-1} (N_r - S_r + 1) \cdot (N_c - S_c + 1) \quad (17)$$

There are an equal number of additions. Note that the number of elements in the sum is independent of the values of Δr and Δc. Therefore, the number of operations is:

$$S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (N_c - S_c + 1) \,. \quad (18)$$

For the second type, First-block, the number of multiplications for each combination is $2 \cdot (N_c - S_c + 1)$, and this is summed over the combinations:

$$\sum_{\Delta r=0}^{S_r-1} \sum_{\Delta c=0}^{S_c-1} 2 \cdot (N_c - S_c + 1) \cdot (E(\Delta r) - 1). \quad (19)$$

There are an equal number of additions. Once again using simple arithmetic, which includes the sum of an arithmetic series, the number of multiplications and additions is:

$$S_c \cdot S_r \cdot (N_c - S_c + 1) \cdot (S_r - 1) \,. \quad (20)$$

The number of multiplications and additions required by the third type, First-per-block, is computed in a similar fashion and is:

$$S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (S_c - 1) \,. \tag{21}$$

For the fourth type, Remaining-indices, we show the number of operations required by all the combinations. As the difference between the number of multiplications and additions is a constant, we use α to denote the constant. Upon completion, α can be substituted with 4 for multiplications (required for a, b, c, d) and 7 (a, b, c, d and the three other operands) for additions.

$$\sum_{\Delta r=0}^{S_r-1} \sum_{\Delta c=0}^{S_c-1} \alpha \cdot (E(\Delta r) - 1) \cdot (B(\Delta c) - 1). \tag{22}$$

This is reduced to:

$$\frac{\alpha}{4} \cdot S_r \cdot S_c \cdot (S_r - 1) \cdot (S_c - 1). \tag{23}$$

Finally we sum (18), (20), (21), (23) and multiply them by two:

$$\begin{aligned} Total = 2 \cdot (&S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (N_c - S_c + 1) + \\ &S_c \cdot S_r \cdot (N_c - S_c + 1) \cdot (S_r - 1) + \\ &S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (S_c - 1) + \\ &\frac{\alpha}{4} \cdot S_r \cdot S_c \cdot (S_r - 1) \cdot (S_c - 1)). \end{aligned} \tag{24}$$

Fig. 6(a) shows the ratio between the number of operations required by the straightforward approach as given in Section 2 and those required by the new approach for both multiplications and additions. This ratio is also indicative of the possible sequential speedup. Note that these curves have the same bell like shape as the straightforward algorithm, Fig. 2. In the Results section, we confirm that the speedups achieved by the new algorithm for both single-core and multi-core follow this curve.

3.4 Additional Implementation Details

In [9], several algorithmic optimizations were presented that are also somewhat relevant to this algorithm. The first is that the results are not accumulated into the final output matrix, but rather to a temporary array before being written to the final output matrix. The motivation for this is that in the older algorithm the indices of a given combination were accessed numerous times. Given that a combination accesses indices on a given diagonal, the older algorithm had a non-sequential access pattern to the output array, causing poor cache performance. By using a temporary sequential array of size $S_r \times S_c$ for each thread, this bad access pattern is avoided. Given that the largest combination writes to $S_r \times S_c$ indices, this is an upper bound on the memory used by a given thread. This increases the memory requirement by a total of $p \times S_r \times S_c$, but improves cache performance. This increase is small compared to the size of the output matrix, $S_r \cdot S_c \times S_r \cdot S_c$. For our new algorithm, this non-sequential access pattern is not an issue, as each index is written to exactly once. The pseudo code in Algorithm 1 assumes that the writing is done to the final output matrix.

The algorithm in [9] computes an optimal number of multiplications. Our new algorithm computes some products more than once - as such the number of multiplications of our new algorithm is not optimal but is within a constant factor of the optimal. The new algorithm can be optimized to compute the optimal number of multiplications at the cost of adding algorithmic overhead, but this is beyond the scope of this paper.

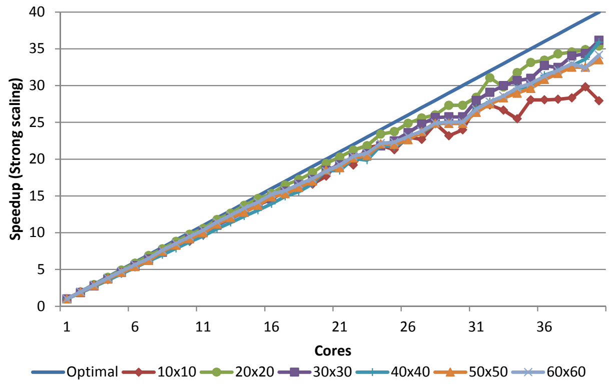


Figure 7: Strong scaling speedup of the new algorithm for multiple window sizes.

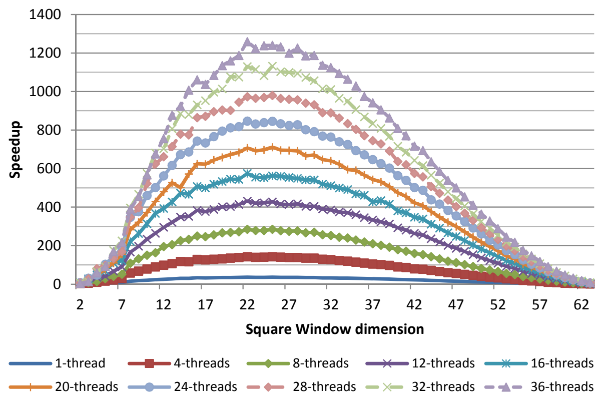
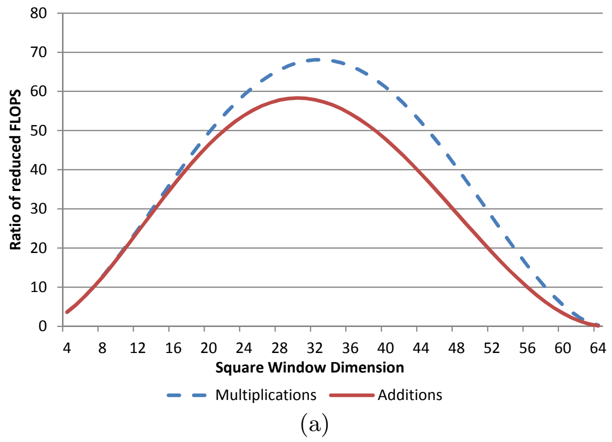


Figure 9: Speedup of the our new algorithm for a 64 × 64 input matrix with square windows over the straightforward algorithm. x-axis is the window size. The speedup curves are in multiple of 4 threads. Note that the curves are equidistance from each other.

4. EMPIRICAL RESULTS

In this section we present performance measurements for the new algorithm, focusing on its scalability to multiple compute cores. We use a a 4-socket Intel multicore system, each containing an Intel Xeon E7-8870 10-core hyperthreaded 2.4GHz processor with a 30MB L3 cache. There are thus 40 physical cores and support up to 80 logical cores. In our tests, we did not use the HyperThreading option. All the algorithms were implemented in C and used OpenMP. The server is equipped with 256 GB of 1066 MHz DDR3 DRAM. Our algorithm was implemented such that intermediate results are stored in dense temporary arrays rather than in the output matrix, thereby increasing cache efficiency.



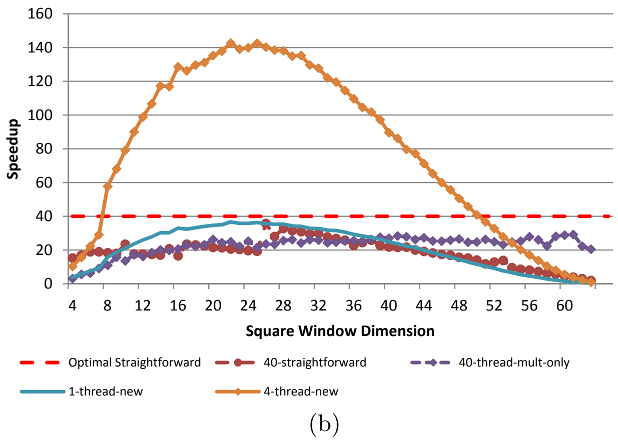


Figure 6: (a) Reduction (factor) in additions and multiplication relative to the straightforward algorithm. (b) Actual speedups based on the execution of three different parallel algorithms (with different thread counts): straightforward, multiplication-reducing only algorithm [6, 9], and the new algorithm. No more than 4 threads are displayed for the new algorithm for figure-scaling purposes. Its speedup scales almost linearly to 40 threads.

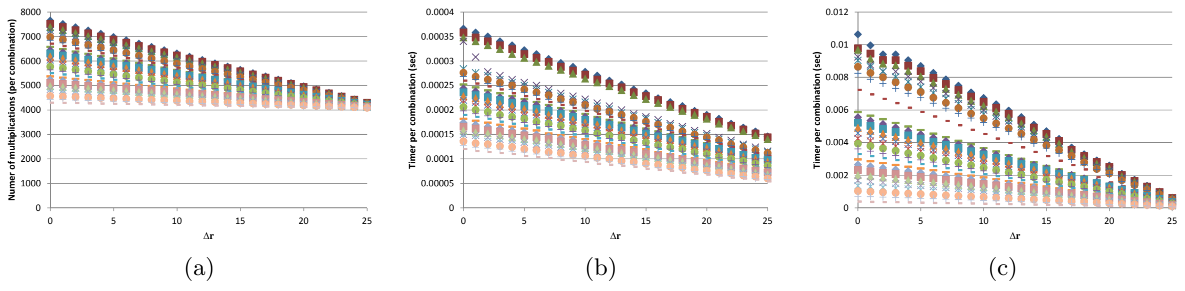


Figure 8: Given a 64×64 input matrix with a 26×26 window: (a) theoretical number of multiplications for each combination of the new algorithm, (b) run times for each combination of the new algorithm, and (c) run times for each combination using the previous combination-based algorithm [6]. There are 676 combinations that are presented in all these sub-figures. Note the different units and scales of the ordinate for the different sub-figures.

We show strong scaling (same total amount of work with an increase of cores) results for the different algorithms relative to their sequential implementation. We also show the performance of these algorithms relative to the sequential implementation of the straightforward dense vector-vector multiplication approach. We implemented the straightforward algorithm using Intel's MKL [1] and received a performance gain from these optimized functions. The MKL library has many optimized kernels for matrix multiplication that use SIMD. The speedup attained from MKL for a single core is approximately 2X for a single thread. For 40 threads there was a very small performance gain of using MKL over our straightforward implementation. All the algorithms including our new algorithm would benefit from SIMD multiplications and additions. These low-level optimizations were outside the scope of this work. For this reason, we compare our new algorithm to the straightforward implementation.

We do our best to report the best possible execution times for these algorithms, and note that all the algorithms benefit from the same compiler optimizations.

Initially, we show that the single core execution of the new algorithm behaves as expected. Fig.6(b) depicts the speedup of the parallel straightforward algorithm and of the new algorithm with several thread counts relative to the sequential straightforward algorithm (dense vector-vector multiplication). The curves indeed reflect the reduction in the number of operations as depicted in Fig.6(a). The figure also depicts the parallelization speedup of the straightforward algorithm. (It is trivially parallelizable because the computations for each index are carried out “from scratch”, not using any partial results.) The maximum possible speedup of this algorithm is 40X (limited by the number of cores), but it is not attained, possibly due to poor cache performance. In fact, the single-core (sequential) execution of the new algorithm outperforms the 40-core execution of the straightforward algorithm. The speedup of the algorithm from [6] is also presented. This algorithm outperforms the straightforward implementation due to a reduction in multiplications and improved access pattern.

Fig. 7 depicts the strong scaling of the new algorithm. Six different window sizes were selected: $10 \times 10, 20 \times 20, 30 \times 30, 40 \times 40, 50 \times 50$, and 60×60. While the strong scaling is not perfectly linear, 85%-90% of maximum system utilization is achieved. The actual run times on the 40-core system are considerably short for the new algorithm, from 0.7ms for the 10×10 window and up to 77ms for the 60×60. With up to 35 cores, the execution times were reasonably consistent for the smaller window sizes. With more cores, carefully timed runs were necessary given that even a brief system call can significantly change the execution times. This was not a problem for the other algorithms as they are considerably slower, so a systems call doesn't significantly affect their timing .

The scalability of the new algorithm is due to its intrinsically balanced load, improved over that of [6]. In Fig. 8(a) we show the number of computations required by the POS combinations of a 26×26 window for a 64×64 input matrix. The x-axis represents the Δr for each of the combinations. For each of Δc, we plotted the number of operations required by the combination. As such, there are 26 curves in the graph, each with 26 points. Note that the ratio between the most computationally demanding combination and the least computationally demanding one is less than 2. This imbalance will only be felt when the number of cores is on the same order of magnitude as the number of combinations. For verification purposes, we timed the computation of each combination; these are depicted in Fig.8(b). Note the 6:1 ratio of actual execution times, larger than the ratio of the number of operations but still moderate, becoming insignificant when there are many more combinations than cores.

Fig. 8(c) depicts the execution times of the combinations for the older algorithm [6]. Clearly, the additions are the bottleneck. Also, the ratio between the most computationally demanding combination (taking 10.6 ms) and the least computationally demanding combination (0.057 ms) is 186:1, which causes load balancing problems for this algorithm [6]. This is due to the greater variability in the number of additions (when carried out from scratch) among combinations than the variability in the number of products. This problem is discussed in depth in [6, 9]. The execution times for the combinations using this algorithm are depicted in Fig.8(c).

Fig. 9 depicts the speedup of the new algorithm versus a square window size. Curves are provided for thread counts in multiples of four. Also, the curves are nearly equally separated, which can be expected in view of the near linear scaling.

Note that the parallel algorithm can achieve a $1200X$ speedup over the straightforward single-core implementation. This is significant for algorithms such as MVM [7], where 10k estimated covariance matrices need to be computed for every image, many times per second if image sequences must be processed. Obviously, these different matrices can be computed in parallel using any of the other algorithms mentioned in this paper. However, these will either increase the memory footprint or require the use of atomic instructions. We have also checked the performance of our algorithm on larger input matrices with varying window sizes, up-to the $32k \times 32k$ image size that was used in [2]. For these tests we always used 40 threads for all the algorithms. For the $1k \times 1k$ inputs matrices speedups were up-to $1121X$. For the larger inputs, the older algorithms timed out, whereas our algorithm completed.

5. CONCLUSIONS

In this paper we presented a new approach for computing the estimated covariance matrix. It is dramatically more efficient than the dense vector-vector multiplication as expressed in the formulation of the Covariance Method. The new approach reduces the total number of floating point operations by almost completely avoiding redundant operations. Also, it requires fewer memory accesses as each datum is used fewer times. All this allows for a faster sequential algorithm - 35X faster than the straightforward algorithm, and 17X faster than the algorithm in [6], which minimizes the required number of multiplications.

The key improvement of the new algorithm over [6], is an incremental approach to computing the partial sums of products of input-matrix element pairs required for computing each output-matrix element. So doing dramatically reduces the number of additions and subtractions, as well as the number of memory accesses.

In addition to saving multiplications, the main contribution of [6] was an elegant partitioning of the output-matrix elements among compute threads, such that no inter-thread synchronization is required. This permits fine-grain parallelism. The shortcoming of [6] is that it did not reduce the number of additions and multiplications, which limited its sequential performance and also, due to a highly variable amount of work for different partitions, created a load-balancing problem for the parallel version.

The new algorithm adopted the partitioning of [6], so it is easily parallelizable. Moreover, the savings in additions and subtractions also sharply reduced the work-variability among partitions, so the parallelism translates more smoothly to high performance. With 40 single-thread compute cores, its parallel version is $1200X$ faster than the single-core implementation of the straightforward algorithm, and some $40X$ faster than [6].

The new algorithm thus apparently dominates the prior art. Moreover, the dramatic performance and efficiency improvements suggest that it may make the Covariance Method more broadly applicable, and in many applications it may be possible to execute it in real time, on mobile platforms, etc., with important impact on the usage mode of those applications.

We note that the new algorithm can also benefit from SIMD optimizations. However SIMD was not within the scope of work which. We encourage others to complete this.

6. REFERENCES

[1] Intel math kernel library, 2013.

[2] S. H. Bokhari, U. V. Catalyurek, and U. V. Gurcan. Massively multithreaded maxflow for image segmentation on the cray xmt2. 2013.

[3] R. Brüschweiler and F. Zhang. Covariance nuclear magnetic resonance spectroscopy. *The Journal of chemical physics*, 120:5253, 2004.

[4] J. Capon. High-resolution frequency-wavenumber spectrum analysis. *Proceedings of the IEEE*, 57(8):1408–1418, 1969.

[5] F. Cayre, C. Fontaine, and T. Furon. Watermarking security: Theory and practice. *Signal Processing, IEEE Transactions on*, 53(10):3976–3987, 2005.

[6] L. David, A. Galperin, O. Green, and Y. Birk. Efficient parallel computation of the estimated covariance matrix. In *IEEE 26th Convention of Electrical and Electronics Engineers in Israel*, 2010.

[7] S. DeGraaf. Sar imaging via modern 2-d spectral estimation methods. *Image Processing, IEEE Transactions on*, 7(5):729–761, 1998.

[8] R. Fante, E. Barile, and T. Guella. Clutter covariance smoothing by subaperture averaging. *Aerospace and Electronic Systems, IEEE Transactions on*, 30(3):941–945, 1994.

[9] O. Green, L. David, A. Galperin, and Y. Birk. Efficient parallel computation of the estimated covariance matrix. *arXiv preprint arXiv:1303.2285*, 2013.

[10] A. Jakobsson, S. R. Alty, and J. Benesty. Estimating and time-updating the 2-d coherence spectrum. *Signal Processing, IEEE Transactions on*, 55(5):2350–2354, 2007.

[11] A. Jakobsson, S. Marple Jr, and P. Stoica. Computationally efficient two-dimensional capon spectrum analysis. *Signal Processing, IEEE Transactions on*, 48(9):2651–2661, 2000.

[12] S. Kay. *Modern spectral estimation*. Pearson Education, 1988.

[13] J. Li and P. Stoica. An adaptive filtering approach to spectral estimation and sar imaging. *Signal Processing, IEEE Transactions on*, 44(6):1469–1484, 1996.

[14] Z. Liu, H. Li, and J. Li. Efficient implementation of capon and apes for spectral estimation. *Aerospace and Electronic Systems, IEEE Transactions on*, 34(4):1314–1319, 1998.

[15] P. López-Dekker and J. Mallorquí. Capon-and apes-based sar processing: performance and practical considerations. *Geoscience and Remote Sensing, IEEE Transactions on*, 48(5):2388–2402, 2010.

[16] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580, 1975.

[17] S. Marple Jr and W. Carey. Digital spectral analysis with applications. *The Journal of the Acoustical Society of America*, 86:2043, 1989.

[18] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast gpu-based ct reconstruction using the common unified device architecture (cuda). In *Nuclear Science Symposium Conference Record, 2007. NSS'07. IEEE*. IEEE, 2007.

[19] E. Yadin, D. Olmar, O. Oron, and R. Nathansohn. Sar imaging using a modern 2d spectral estimation method. In *Radar Conference, 2008. RADAR'08. IEEE*, pages 1–6. IEEE, 2008.