ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/235892378

Efficient Parallel Computation of the Estimated Covariance Matrix

Conference Paper · March 2013

DOI: 10.1109/EEEI.2010.5661930 · Source: arXiv

CITATIONS
2

READS
122

4 authors, including:

Oded Green
NVIDIA
41 PUBLICATIONS **570** CITATIONS

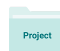
SEE PROFILE

Yitzhak Birk
Technion - Israel Institute of Technology
94 PUBLICATIONS **2,049** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Hashing View project


High Performance Graph Analytics View project

All content following this page was uploaded by Oded Green on 15 August 2014.

The user has requested enhancement of the downloaded file.

Efficient Parallel Computation of the Estimated Covariance Matrix

Lior David, Ami Galperin, Oded Green and Yitzhak Birk
Technion, Israel

***Abstract*—Computation of a signal's estimated covariance matrix is an important building block in signal processing, e.g., for spectral estimation. It involves a sliding window over an input matrix, and the summation of products to construct any given output-matrix element. Any given product contributes to multiple output elements, thereby complicating parallelization. We present a novel algorithm that attains very high parallelism without repeating multiplications or requiring inter-core synchronization. Key to this is the assignment to each core of distinct diagonal segments of the output matrix, selected such that no multiplications need be repeated, and exploitation of a shared memory (including L1 cache) that obviates the need for a corresponding awkward partitioning of the memory among cores. Implementation on Plurality's shared memory many-core architecture and, in order to demonstrate additional benefits, also on the x86, reveals linear speedup and a 130-fold power-performance advantage over x86.**

***Index Terms*—Parallel algorithms, Parallel processing, Synthetic aperture radar, Spectral analysis, Radar signal processing, Estimation, Covariance estimation.**

I. Introduction

Covariance estimation is widely used for signal processing and even for cryptanalysis. Sliding window averaging (aka "sub-aperture averaging" and "the covariance method") is currently a prominent autocorrelation (and covariance) estimator [1].

Computation of the estimated covariance matrix essentially entails the averaging of inner products of a sliding window over the input matrix: for each window position within the input matrix, form a vector by column-stacking its elements, and multiply it by its conjugate transpose. Averaging those results over all possible positions of the window within the input matrix yields the estimated covariance matrix.

The product of any two input-matrix elements contributes to multiple output-matrix elements, yet matrices may be large and the number of different two-element products may be huge, so efficient computation on a real system poses apparently conflicting challenges: 1) computational efficiency (specifically, avoiding repetition of multiplications) and 2) efficient use of memory (both reducing the memory footprint and improving locality, the latter for efficient use of caches). For parallel implementations, additional challenges include 3) partitioning the work into many pieces and 4) prevention of contention and synchronization requirements among compute cores.

In this paper we present a novel, efficient and highly parallel way to compute the estimated covariance matrix, jointly addressing all the aforementioned challenges. (The results remain unchanged.) Key to our success is a unique partitioning of the output matrix elements among compute tasks, combined with the use of a shared memory (including L1 cache) many-core architecture (by Plurality Ltd.) to circumvent the resulting awkward memory partitioning.

Plurality's HAL architecture [2] features tens to hundreds of compute cores, interconnected to an even larger number of memory banks that jointly comprise the shared cache. The connection is via a high speed, low latency combinational interconnect. By so doing, one enjoys the benefits of a uniform memory architecture without suffering from the communication bottleneck of a shared bus. Also, memory coherence comes free, as there is no private memory. The memory hierarchy includes off-chip (shared) memory, which was not required in our case. Finally, the programming model is a set of sequential "tasks" along with a set of precedence relations among them, and these are enforced by a very high throughput, low latency synchronizer/scheduler that dispatches work to the cores.

Our focus has been on parallelization. Nonetheless, some of the elements of our approach also improve the efficiency of sequential implementations, as will be pointed out later.

The remainder of this paper is organized as follows. Section II formulates the problem. Section III describes the new parallel algorithm. Section IV provides experimental results and section V summarizes the work.

II. The Covariance Method

A. Background

The use of estimated covariance matrices originated from the area of speech processing [3]. One way to compute the estimated covariance matrix is by the Covariance Method. The method's main rationale is minimizing the error in the estimate of a covariance matrix of a time series. Good estimates can give insights about the data periodicities and enable fast and accurate analysis of the input data.

The method can be used for signal processing algorithms, e.g., whenever the correlation between signal history domain data samples is needed (as in [4]), to smooth spatial clutter by averaging over given transposed data (see [5]) or perform 2D spectral analysis (e.g., [6]). In all the aforementioned cases, the computation of the estimated covariance matrix is an important computational building blocks.

The covariance method is a biased estimator, and its output is a Hermitian, positive semi definite matrix, so it is guaranteed

978-1-4244-8682-3/10/$26.00 © 2010 IEEE

to be non-singular. This is unlike the output of other methods, such as autocorrelation, and often causes the covariance method to be preferred [1].

B. The Serial Algorithm

Terminology and Symbols: All indices (rows and columns) start at 1, i.e., the first element in $\vec{V}$ is $\vec{V}(1)$, which is also denoted $\vec{V}_1$. The input matrix is denoted A, with N rows and M columns. $S^{r,c}$ represents the sliding $P \times Q$ window, where the (r, c) superscripts denote the position of its upper left corner. $\vec{V}^{r,c}$ is the column stack of $S^{r,c}$ such that

$$S^{r,c}(i, j) \equiv \vec{V}^{r,c}(P \cdot (j - 1) + i). \tag{1}$$

The conjugate transposes of matrices and vectors are denoted by A^H and $\vec{V}^H$, respectively, and C denotes the algorithm's output matrix. For clarity of exposition, we will refer to *elements* of the input matrix, while those of the output matrix will be referred to as *indices*.

Formulation: The serial algorithm can be formulated as

$$C = \sum_{p=1}^{N-P} \sum_{q=1}^{M-Q} C^{p,q} = \sum_{p=1}^{N-P} \sum_{q=1}^{M-Q} \vec{V}^{p,q} \cdot (\vec{V}^{p,q})^H. \tag{2}$$

Note that C is Hermitian, being the sum of Hermitian matrices, so only the upper triangle or the lower triangle needs to be computed.

C. Deficiencies of a Straightforward Serial Implementation

Based on (2), the multiplication of $\vec{V}^{p,q}$ by $(\vec{V}^{p,q})^H$ is actually a multiplication of every element in each vector by every element in the other vector. Therefore, when considering two sliding window positions $S^{p,q}$ and $S^{p',q'}$ that both contain products with values x and y, both result matrices $C^{p,q}$ and $C^{p',q'}$ will contain the following products as indices: $x \cdot \bar{x}$, $x \cdot \bar{y}$, $\bar{x} \cdot y$, $y \cdot \bar{y}$. However, the relative (to the upper left corner of the window) positions of the products will not be the same in the two matrices.

In Figure 1, for example, the product $A_{3,3} \cdot \overline{A_{4,4}}$ is needed for every window that contains those two elements. It would be desirable to compute it once and then write it to the correct place for each of the windows. (The challenge is to do so without consuming much memory for temporary results or requiring inter-task synchronization.)

D. Parallelization Challenges

Parallelization of the serial algorithm presents several challenges: 1) avoiding redundant multiplications, 2) obviating the need for synchronization and atomic instructions, 3) holding down the required amount of memory for intermediate results and 4) utilizing all cores all the time (massive parallelism with load balancing). 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 task would result in redundant multiplications. Simplistically assigning any given product (multiplication) to a single task would result in multiple tasks

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$	...	$A_{1,M}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$	...	$A_{2,M}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$	...	$A_{3,M}$
$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$	...	$A_{4,M}$
$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$	...	$A_{5,M}$
...	...	...	...	...	...	...
$A_{N,1}$	$A_{N,2}$	$A_{N,3}$	$A_{N,4}$	$A_{N,5}$	...	$A_{N,M}$

Fig. 1. Use of a product. The elements $A_{3,3}$ and $A_{4,4}$, and consequently the product $A_{3,3} \cdot \overline{A_{4,4}}$, are contained in several sliding window positions. The presented positions are all the borderline ones, such that moving a window one step in the directions "up" (yellow, green), "down" (blue, red), "left" (yellow, red), "right" (green, blue) would result in the product no longer being in the window.

contributing to the same element of the output matrix, thus requiring synchronization and possibly atomic operations.

As another example, if each concurrent thread keeps a temporary copy $C^{p,q}$ of the estimated covariance matrix, then a great deal of parallelism can be achieved in the multiplication stage, but in the subsequent summation stage most of the cores will not be utilized. Also, this approach requires sizable memory, larger than most caches, and the resulting cache misses would hurt performance. Finally, this approach does not avoid redundant multiplications.

We next present our novel parallel algorithm, which jointly addresses all the aforementioned challenges.

III. The Parallel Algorithm

A. Multiplication Combinations

In the previous section, it was shown that the product of any given pair of matrix elements may belong to several window positions. Also, the elements' positions relative to each other are the same regardless of the window. However, the product of any element pair in the context of each of the windows containing it will be written to a different index in the output matrix based on the position of the elements in the window (which is responsible for the creation of $\vec{V}^{p,q}$). We next proceed to show how efficient parallelization can be achieved despite this complexity.

Definition. Let A_{r_1,c_1} and A_{r_2,c_2} be two input-matrix elements. Their *inter-element distance* is defined as $\Delta \triangleq (\Delta r, \Delta c) = (r_2 - r_1, c_2 - c_1)$.

Definition. A *multiplication combination*, *combination* for

$C_{1,1}$	$C_{1,2}$	$C_{1,3}$	$C_{1,4}$	$C_{1,5}$	$C_{1,6}$	…	$C_{1,PQ}$
$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{2,4}$	$C_{2,5}$	$C_{2,6}$	…	$C_{2,PQ}$
$C_{3,1}$	$C_{3,2}$	$C_{3,3}$	$C_{3,4}$	$C_{3,5}$	$C_{3,6}$	…	$C_{3,PQ}$
$C_{4,1}$	$C_{4,2}$	$C_{4,3}$	$C_{4,4}$	$C_{4,5}$	$C_{4,6}$	…	$C_{4,PQ}$
$C_{5,1}$	$C_{5,2}$	$C_{5,3}$	$C_{5,4}$	$C_{5,5}$	$C_{5,6}$	…	$C_{5,PQ}$
$C_{6,1}$	$C_{6,2}$	$C_{6,3}$	$C_{6,4}$	$C_{6,5}$	$C_{6,6}$	…	$C_{6,PQ}$
…	…	…	…	…	…	…	…
$C_{PQ,1}$	$C_{PQ,2}$	$C_{PQ,3}$	$C_{PQ,4}$	$C_{PQ,5}$	$C_{PQ,6}$	…	$C_{PQ,PQ}$

Fig. 2. Output Matrix. Indices of different colors are written by different combinations.

short, is an inter-element distance $(\Delta r, \Delta c)$ that satisfies

$$\begin{aligned} -(P-1) &< \Delta r < P-1 \\ -(Q-1) &< \Delta c < Q-1. \end{aligned} \tag{3}$$

Each combination corresponds to a single distance value. (The restrictions on the distances reflect the fact that we are only interested in products of elements within the same window.)

Note that the combinations are based on the relative position of the elements. For example, the element pairs $(A_{3,3}, A_{4,4})$ and $(A_{5,5}, A_{6,6})$ (and the respective products) belong to the same combination.

B. Parallelization Highlights

We begin with a critical insight whose proof is omitted for brevity: any index in the output matrix is the sum of products, all of which belong to the same combination.

This, along with the fact that the sets of element pairs belonging to different combinations are disjoint, implies that assigning each combination to a different task permits concurrent execution of the tasks with no need for synchronization among them and with no memory-write conflicts. Also, the product of any two elements is computed at most once.

Another critical insight, whose proof is also omitted for brevity, is that the target indices of any given combination form a diagonal segment (in the output matrix). More details on this will be provided later.

Next, recall that the estimated covariance matrix is Hermitian, so it suffices to compute the upper triangle or lower triangle sub-matrices. Also, any two inverse-distance combinations (e.g. $(\Delta r, \Delta c) = (1, 1)$ and $(\Delta r, \Delta c) = (-1, -1)$) consist of the same element-pair values. As the multiplication is done by taking the *conjugate* of one of the elements, we can utilize the trivial identity $A_{x',y'} \cdot \overline{A_{x,y}} \equiv \overline{A_{x,y} \cdot \overline{A_{x',y'}}}$ to further reduce the number of actual multiplications.

Finally, note that any given diagonal segment resides entirely within one of the two triangles or in the main diagonal of the output matrix. Consequently, the products belonging to combinations that "own" the lower triangle needn't be computed. Accordingly, the set of *unique combinations* is specified by mofifying the distance restrictions of (3) as follows:

$$\text{Unique Combinations} \triangleq \text{each } (\Delta r, \Delta c) \text{ in}$$
$$\begin{Bmatrix} 0 \le \Delta r \le P-1 \\ 0 \le \Delta c \le Q-1 \end{Bmatrix} \cup \begin{Bmatrix} -(P-1) \le \Delta r \le -1 \\ 1 \le \Delta c \le Q-1 \end{Bmatrix} \tag{4}$$

As can be seen, Δc is always positive, so the second element is to the right of the first one. Δr, in contrast, can be positive or negative, so there is no keen observation to make on its value.

C. Additional Combination Properties

We now offer a more detailed illustration of the relationship between window position and the target index of the product of a single element pair contained in the window; i.e., a multiplication centric description rather than a window centric one.

Definition. Given two elements in the input matrix A_{r_1,c_1} and A_{r_2,c_2}, the *initial placement* of the sliding window for those two elements is its lowest and rightmost location such that the window contains those two elements and is entirely contained within A. For convenience, we denote the initial placement by the location of its upper left corner.

The sliding window may only be moved from its initial position to the left and upward (see Figure 3b and Figure 3c). Assuming that a product is written to $C_{r,c}$ when the window is in its initial position (see Figure 3d), two important observations are made in Figure 3: 1) sliding the window one step to the left (Figure 3b) results in writing the product to $C_{r+P,c+P}$ (Figure 3e), and 2) sliding the window one step upward (Figure 3c) results in writing the product to $C_{r+1,c+1}$ (Figure 3f).

Based on the above, several important observations can be made: 1) for elements located near the boundaries of A, it may not be possible to move the sliding window as it might exit the boundaries of A; accordingly, some multiplications will not write to all their designated result indices; 2) the total number of legal positions of the sliding window about a given multiplication (matrix-element pair) is up to $(P-|\Delta r|) \cdot (Q-|\Delta c|)$. We next prove this.

Given the two elements and their initial placement, it is possible to move the sliding window around them leftward and upward, as can be seen in Figure 3b and Figure 3c. The maximum number of times that a sliding window can be moved to the left while staying within the bounds of the matrix A is $(P-|\Delta r|)$. Next, the sliding window is returned to its initial position (Figure 3a) and is then moved upward by one position. Next, it is again possible to move the sliding window a total of $(P-|\Delta r|)$ steps to the left. This is repeated a total of $(Q-|\Delta c|)$ times, so there are $(P-|\Delta r|) \cdot (Q-|\Delta c|)$ legal positions for any given "element-pair centric" sliding window.

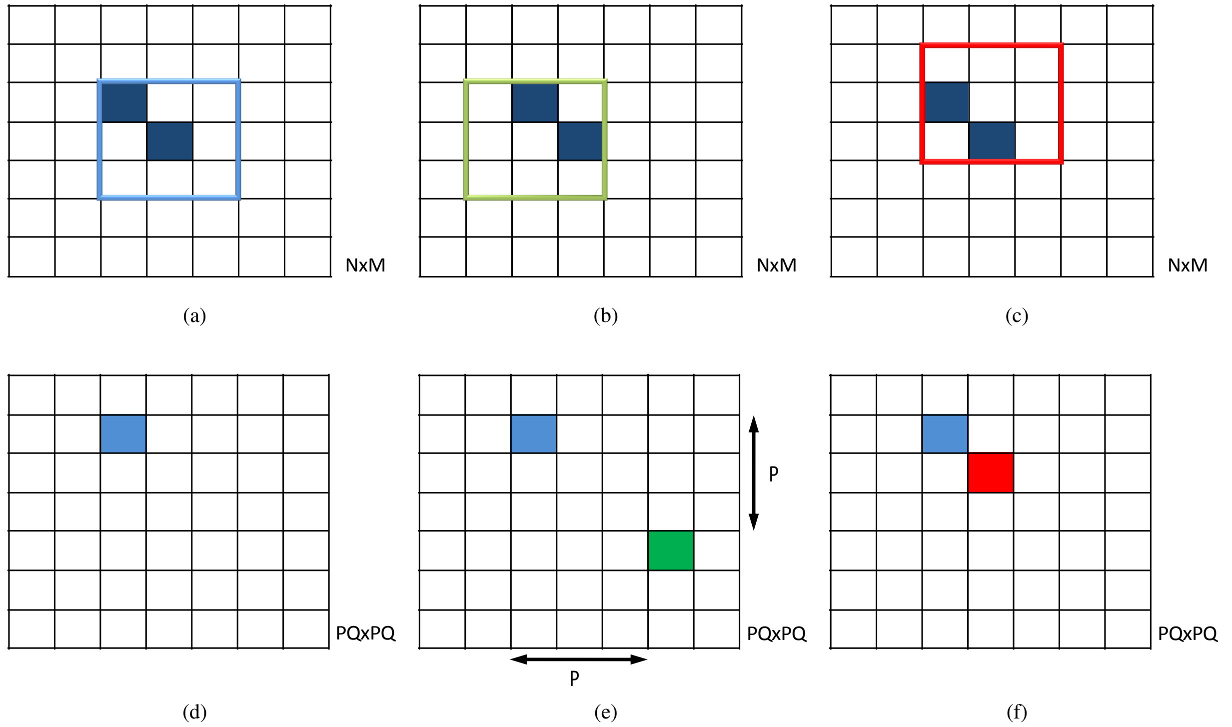


Fig. 3. The sliding window over an element pair. (a) Initial placement of the sliding window in the input matrix, surrounding the two elements; (b) moved one step leftward; (c) moved from its original position one step upward; (d) The output-matrix index to which the product is written in the context of the (a) window position; (e) the index to which the same product is written in the (b) positioning context; (f) the index to which the same product is written in the (c) positioning context.

An implication of these observations is that the number of consecutive indices on a given diagonal that will be written to by a given combination depends on the number of times that the window can be moved upward, while the number of times that the window can be moved leftward will determine the number of disjoint segments that a given combination writes to on the given diagonal. For certain multiplications (mainly those that are near the border of A), the result will not be written to all of the combination's designated indices.

We next present the actual algorithm.

D. An Efficient Parallel Algorithm

Because each combination writes to a distinct set of output-matrix indices, as can be seen in Figure 2, it is possible to compute the different combinations concurrently on multiple cores without allocating temporary matrices. For a shared-memory architecture in which the cache size is limited, this is very important. Furthermore, as each combination is the only one to access its indices, there is no need for locks, synchronization or even atomic instructions. Last but not least, as each combination is responsible for computing multiplications that are part of the combination and writing the results, each unique product is calculated only once.

Partitioning by combinations, as presented in algorithm 1, thus requires the bare minimum number of multiplications. This is a dramatic reduction relative to the naive approach.

Algorithm 1: The Parallel Algorithm

Data: A - the $N \times M$ input data matrix.
Result: C - the estimated covariance $P \times Q$ matrix.

```
1  begin
2      foreach unique combination (Δr, Δc) do in parallel
3          foreach possible position of S in A do
4              d ← A(r_1, c_1) · conj(A(r_2, c_2))
5              compute initial placement of S
6              foreach (r_{l,u}, c_{l,u}), a valid shift of S do
7                  C(r_{l,u}, c_{l,u}) ← C(r_{l,u}, c_{l,u}) + d
8              end
9          end
10     end
11 end
```

As combinations are mutually independent and their destination indices are disjoint, it is possible to execute them concurrently on different cores with no need for synchronization. Whenever a core becomes available, a combination may simply be allocated to it. Thus, parallelism is achieved at the combination granularity.

IV. Results

The new algorithm was implemented and tested on Plurality's HAL shared memory many-core architecture and on Intel

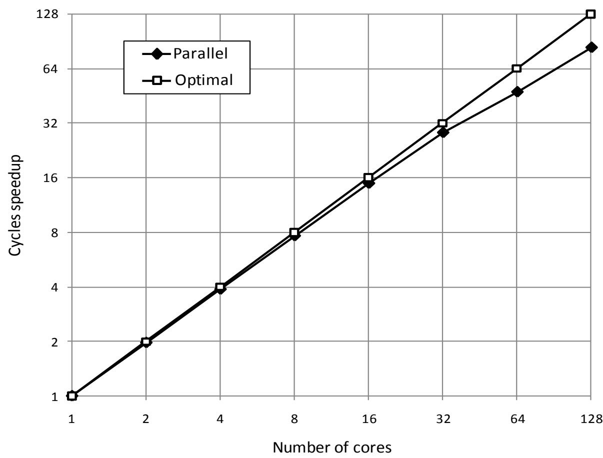


Fig. 4. Speedup vs. number of cores. This simulation was done on Plurality's simulator, for a 32x32 input matrix and a 13x13 window. Speedup with 128 cores is 84X.

TABLE I
COMPARISION BETWEEN X86 AND THE PLURALITY PLATFORMS

	IPC[1]	Freq. [GHz]	Power [W]	No. of Cores
Plurality[2]	1	0.4	4	64
Core 2 Duo[3]	1.0086	2.4	65	1

[1] Instructions Per Clock [2] see [2] [3] results by [7]

Core 2 Duo. For further details, see Table I. We begin with results for the HAL platform.

One of HAL's strongest features is its efficient hardware scheduler that allocates tasks in a very small number of clock cycles, rendering dispatch overhead negligible. This, combined with the large number of combinations and the fact that any core can do any job with equal efficiency (due to the uniform memory architecture and absence of private caches), enables near-linear speedup, as depicted in Figure 4.

As the number of cores increases, the speedup becomes less than perfect due to the unequal workload of the combinations. This can be dealt with in various ways, and details are beyond the scope of this paper. It should moreover be noted that oftentimes multiple covariance matrices are to be computed. In these cases, and in view of the efficient use of memory, they can be computed concurrently, naturally resulting in better load balancing among the cores. We have seen perfect linear speedup with 256 cores.

We now turn to a comparison between the HAL and Intel x86 platforms, focusing on power-performance ratio in Table I. The Plurality HAL performance results are based on cycle-accurate simulation and the chip's estimated power consumption. Those for the Intel x86 core 2 duo are based on an optimistic estimate of performance (IPC of 1.3 instead of 1.0086, and a high cache hit rate) and on power consumption data. With this,

$$\frac{\text{Plurality}}{\text{Core 2 Duo}}\left[\frac{\text{performance}}{\text{power}}\right] = \frac{\frac{\text{freq}\cdot\text{IPC}\cdot\#\text{ cores}}{\text{power}}}{\frac{\text{freq}\cdot\text{IPC}}{\text{power}}} = 133.3. \quad (5)$$

In view of the aforementioned assumptions, the estimated 133X advantage of HAL is thus conservative.

The new algorithm was also implemented on a single core of the Intel Core 2 Duo from Table I. The results show an improvement in execution time relative to the straight-forward implementation of the serial algorithm. This can be expected from the savings in multiplications, but is not obvious for two reasons: 1) potentially reduced memory locality (the parallel algorithm was designed for a several-Megabyte shared-cache machine), and 2) the overhead of a parallel algorithm. Further details of this are beyond the scope of this paper.

V. CONCLUSIONS

This paper presented a novel approach for parallelizing the computation of an estimated covariance matrix, an important building block for digital signal processing. Using critical insights pertaining to the relationship between input-matrix elements and output-matrix indices to which they contribute, efficient parallelization was made possible: no multiplications are repeated, yet no coordination is required among cores and the memory footprint is very small. In so doing, we took advantage of a unique share-memory architecture that naturally supports an otherwise awkward partitioning of memory among cores.

Experimental results show near perfect speedup on tens of cores for a single matrix, and perfect linear speedup on as many as 256 cores when several matrices are computed concurrently. Finally, some of the insights and corresponding approaches are relevant even to single-core implementations and sequential execution.

While the paper focused on a particular computation, the insights and approaches are likely to be broadly applicable.

ACKNOWLEDGMENTS

The authors are Grateful to Oz Shmueli of the Parallel Systems Lab in the Electrical Engineering department for his assistance.

REFERENCES

[1] S. L. Marple, *Digital Spectral Analysis With Applications*. Prentice Hall, 1987.
[2] Plurality's website. [Online]. Available: http://www.plurality.com
[3] J. Makhoul, "Linear prediction: A tutorial review," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 561–580, apr. 1975.
[4] S. DeGraaf, "Sar imaging via modern 2-d spectral estimation methods," *Image Processing, IEEE Transactions on*, vol. 7, no. 5, pp. 729–761, may 1998.
[5] R. Fante, E. Barile, and T. Guella, "Clutter covariance smoothing by subaperture averaging," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 941–945, jul. 1994.
[6] A. Jakobsson, J. Marple, S.L., and P. Stoica, "Computationally efficient two-dimensional capon spectrum analysis," *Signal Processing, IEEE Transactions on*, vol. 48, no. 9, pp. 2651–2661, sep. 2000.
[7] T. K. Prakash, "Performance Analysis of Intel Core 2 Duo Processor," Master's thesis, Louisiana State University, 2007.

View publication stats