Optimizing Space Time Adaptive Processing Through Accelerating Memory-bounded Operations

Tze Meng Low
Department of Electrical and Computer Engineering
Carnegie Mellon University
Email: lowt@cmu.edu

Qi Guo
Department of Electrical and Computer Engineering
Carnegie Mellon University
Email:guoqi@cmu.edu

Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
Email: franzf@cmu.edu

***Abstract*—Space-Time Adaptive Processing (STAP) is a technique for processing signals from multiple antenna elements over multiple time periods for target detection. As STAP algorithms are typical run on airborne platforms, they need to be both high performance and energy-efficient. Due to the high rate of processing required, many existing algorithms focus on reducing the dimensionality of the data, or exploiting structure in the underlying mathematical formulation in order to reduce the total number of floating-point operations (FLOPs), and consequently, the time for computation. While such algorithms target the FLOPs-intensive operations within the STAP algorithm, a significant portion of the compute time for most STAP algorithms is actually spent in low-FLOPs, memory-bounded operations. In this paper, we address the computation of these memory-bounded operations within the STAP algorithm using a 3D stacked Logic-in-Memory system. The imminent arrival of 3D stacked memory makes avail high memory bandwidth, which opens up a new and orthogonal dimension for optimizing STAP algorithms. We show that more than 11x improvement in time, and 77x improvement in energy efficiency can be expected when a 3D stack is used together with memory-side accelerators to target the memory-bounded operations within STAP.**

I. Introduction

Space-time adaptive processing (STAP) is typically performed on airborne platforms to detect targets from radar signals that contain clutter due to natural or artificial sources (jammers). The general principle of STAP is that we want to test the hypothesis that there exists a target at a particular location, traveling at a particular relative velocity. By adaptively building a filter that enhances the signal in that particular range and velocity, while attenuating the noise, a target is detected if the resulting signal remains above a particular threshold [1]. As multiple possible targets at multiple possible ranges need to be identified in real-time, STAP is inherently a computationally intensive operation. As such, many STAP algorithms have been developed to reduce the computational cost of STAP to make it practical for deployment. Typically, these algorithms improve the performance of STAP by reducing the dimensionality of the data. A summary of these commonly used low-computational load STAP algorithms are described in [2].

In this paper, we focus on an orthogonal approach to optimizing STAP algorithms. We optimize STAP algorithms by reformulating memory-bounded operations into compute-bounded operations. This reformulation of memory-bounded vector operations into compute-bounded matrix operations can be implemented completely in terms of parallel loops around subroutine calls to highly efficient libraries such as Fastest Fourier Transform in the West (FFTW) [3] and Intel's Math Kernel Library (MKL) [4]. The use of subroutine calls to standardized application programming interfaces (APIs) also allows one to easily port the STAP algorithms onto different platforms, including those that utilize new/future technology such as our proposed accelerated 3D stacked memory. For memory-bounded operations that cannot be reformulated, we accelerate their execution on a proposed accelerated 3D stacked memory. We designed a 3D stacked Logic-in-Memory (LiM) system consisting of a multicore processor and 3D stacked DRAM equipped with memory-side accelerators for various memory-bounded operations. As such accelerators are built upon 3D stacked DRAM with high internal bandwidth, the resulting speedup achieves up to 6x compared against the conventional hardware implementation [5].

A. Motivation

We motivate the need for this orthogonal approach to optimizing STAP with the following example. In Figure 1, we show the breakdown of the number of complex floating point operations (FLOPs), and the compute time of the different stages for the third-ordered Doppler-factored STAP algorithm from the PNNL PERFECT Suite [6]. Three separate data configurations (shown in Table I) were tested.

Notice that close to or more than 50% of the time (right stacked columns) is spent in the Normalize and Apply Weights stage of STAP for all three configurations. However, this stage accounts for no more than 44% of the total number of FLOPs in the STAP algorithms. This stage, though low in FLOPs, is computed primarily with many inner-product operations which are inherently memory-bounded. As current memory bandwidth between main memory and the processor is limited, the processor is mostly stalled, waiting for data to be brought in from memory to be processed. Hence it is important to optimize and accelerate memory-bounded operations to attain better computational and power efficiency.

Contributions. In this paper, we make the following contributions:

Orthogonal optimization domain: We optimize STAP algorithms via an approach that is orthogonal to current approaches. By casting operations over vectors into operations on matrices, we turn memory-bounded operations into compute-bounded operations, which can be more efficiently imple-

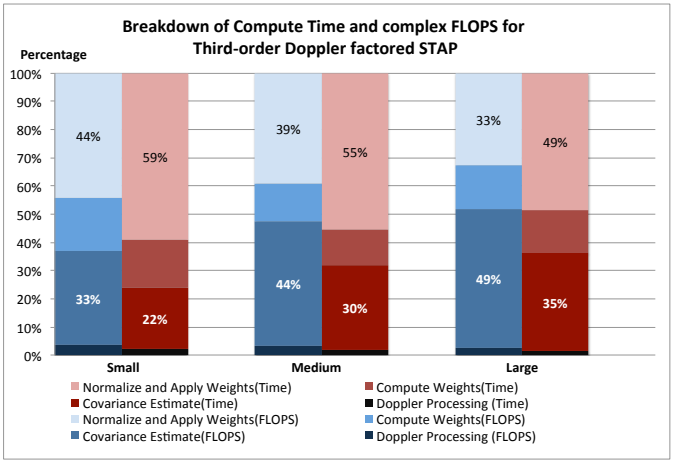

Fig. 1. Breakdown of the compute time and number of complex FLOPs for different configurations provided by the PNNL PERFECT Suite. The left stacked bar shows breakdown of the complex FLOPs, while the right shows breakdown of compute time.

Parameter	Variable	Small	Medium	Large
Spatial channels/ elements	L	4	6	8
Pulses per CPI	P	128	128	128
Doppler bins	K	256	256	256
Range bins	N	512	1024	4096
Range bins per training block	N_R	32	64	64
Temporal degrees of freedom	T_{DOF}	3	3	3
Steering vectors	D	16	16	16

TABLE I. DIFFERENT STAP CONFIGURATIONS FROM PNNL PERFECT SUITE.

mented on current architecture with a hierarchical memory. We utilize a 3D stacked memory with memory-side accelerators to compute memory-bounded operations that cannot be converted into compute-bound operations. These optimization/acceleration can result in more than a magnitude improvement in performance. An additional benefit of this approach is that it allows one to implement the entire STAP algorithm as parallel loops around library calls, thereby increasing the portability of the STAP implementation across a variety of platforms.

Accelerated 3D stacked memory for resource-constraint platforms: We show that 3D stacked memory with memory-side accelerators can significantly improve the performance (in terms of both time and energy) of algorithms on resource-constraint platforms. Memory-side acceleration reduces both the time and energy required (close to 3x and 8x improvement over a highly optimized and parallelized implementation) to transfer data from memory to the processor, making it even more suitable for resource-limited platforms on which STAP algorithms are run. In addition, the memory-side accelerators we introduced to the 3D stacked memory are mapped to the library APIs, and thus, can be reused as computational kernels in other applications.

II. SPACE TIME ADAPTIVE PROCESSING ALGORITHM

In this section, we present a brief overview of the mathematical operations that are performed in the third-order Doppler-factored STAP algorithm implemented in the PNNL STAP benchmark. In this discussion, we assume that the steering vectors, or vectors that model the expected return signal from the target are given as inputs and thus not discussed.

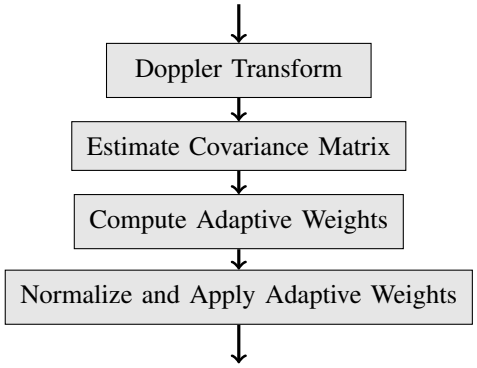

Fig. 2. 4 stages of a post-Doppler STAP algorithm. The first stage transforms the received signal to the Doppler frequency, while last three stages of the algorithm is similar if not identical to many other STAP algorithms.

A. Stages in STAP

A post-Doppler STAP algorithm typically comprises of the four main stages, as shown in Figure 2. The received input is first converted into Doppler frequency via a series of FFTs. Then, STAP is performed in the following three stages: 1) Compute an estimate of the interference covariance matrix $\hat{R}_d$ for the particular range, 2) An adaptive weight, w is computed with the covariance matrix $\hat{R}_d$, and the steering vector v representing the target, and 3) w is normalized and applied to the range data to compute the desired output. We discuss the different stages in detail below:

Doppler Processing: The received signal is first processed by the application of $L \times N$ Discrete Fourier Transforms (DFT), where L is the number of array elements and N is the number of range cells. Typically, as the elements for each DFT are often received in non-consecutive memory addresses, the input data will require a data reshape to reorder them into consecutive memory addresses. This is to improve the performance of each individual DFT. Similarly, because the outputs of the DFTs are not in the form required by subsequent stages, another data reshape is performed after all DFTs have completed. It is important to note that these reshape contains no computation and are purely memory-bounded operations.

Covariance Matrix Construction: For each range cell, an interference covariance matrix is computed. In practice, the actual interference covariance matrix is not known *a priori* and an estimate (denoted $\hat{R}_d$) of it has to be computed in real-time. Typically, N_R range cells around the desired range cell are used to estimate $\hat{R}_d$ in the following manner:

$$\hat{R}_d = \frac{1}{N_R} \sum_{k=0}^{N_R-1} s_k s_k^H, \quad (1)$$

where each $s_k s_k^H$ represents the interference covariance matrix at the k^{th} range cell.

Computing Adaptive Weights: Having computed an estimate of the interference covariance matrix for a particular range, the weight that will be used to build a filter for identifying a target represented by the steering vector v can be obtained by computing

$$w = \hat{R}_d^{-1} v.$$

In practice, $\hat{R}_d$ is not inverted. Instead, $\hat{R}_d$ is factorized either by the QR decomposition or Cholesky factorization, and forward and back-substitutions are performed. Mathematically, this is represented as

$$w = U^{-1} U^{-H} v, \tag{2}$$

where $\hat{R}_d = U^H U$, and U is the Cholesky factor of $\hat{R}_d$.

Applying adaptive weights: Often the adaptive weight vector w is normalized before being used to compute the comparison statistic ψ from which the presence of the target is determined. This normalizing and applying of the adaptive weight w to compute the ψ can be described mathematically by

$$\psi = \frac{w^H s_k}{w^H v} \tag{3}$$

The output ψ is then compared to a predetermined threshold to determine if the target represented by v is present in that particular range represented by the interference covariance matrix constructed with s_k.

III. Optimizing Memory-bounded Operations

Notice that in the description of the STAP algorithm, Equations 1 to 3 are all operations on vectors. As vector operations are memory-bounded operations, i.e. data cannot be brought from memory at a high enough rate, this results in stalls during computation. As such, they inherently cannot be implemented efficiently on current architectures with the conventional memory hierarchy. In this section, we discuss the two methods we used to optimize these memory-bounded operations.

A. From Memory-bounded to Compute-bounded operations

A key observation made by the linear algebra community is that by casting operations into different forms of matrix multiplication, a compute-bounded operation, high performance implementations can be realized on architecture with hierarchical memory. This insight underlies the shift from Level 1 and Level 2 Basic Linear Algebra Subprograms (BLAS) [7], [8] to the Level 3 BLAS [9], and the use of blocked (tiled) algorithms in higher level linear algebra operations (e.g. QR and Cholesky factorizations) in LAPACK [10]. Using the same approach, the first two stages of the STAP algorithm can be converted into compute-bounded operations.

Recall that the first stage of the problem is the computation of the interference covariance matrix given by Equation 1. However, by defining a matrix S as a collection of vectors s_k, i.e.

$$S = [s_0 \; s_1 \; \ldots \; s_{N_R - 1}],$$

Equation 1 can be rewritten as a matrix multiplication of the form:

$$\hat{R}_d = \frac{1}{N_R} S S^H,$$

which is known as the Symmetric Rank-k Update (SYRK) operation in the Level 3 BLAS.

Similarly, the computation of the adaptive weights w (Equation 2) can also be converted to using Level 3 BLAS. We first use the function `potrf` in LAPACK, a highly optimized Cholesky factorization implementation, to compute the Cholesky factor, U, of $\hat{R}_d$. The resulting matrix U is then used to compute a weight w_i for each steering vector v_i. By collecting the steering vectors v_i into a matrix V, Equation 2 can be transformed into the following operation sequence:

$$\begin{aligned} W &= U^{-H} V \\ W &= U^{-1} W \end{aligned}$$

where W is the collection of weights for the collection of vectors, and can be implemented with two calls to the Triangular Solve with Multiple Right-hand Sides (TRSM) operation in Level 3 BLAS.

B. Remaining Bottleneck

By performing the above transformations, the computation of the covariance matrix, and the computation of the adaptive weights can be implemented as loops around either Level 3 BLAS or LAPACK subroutine calls, thereby eliminating all memory-bounded operations in these stages. The remaining four memory-bounded operations are the data reshape, DFTs, inner-product and vector scaling operations in the first and last stages. For these two stages, we rely on the use of accelerators on our 3D stacked Logic-in-Memory system that will be discussed in the subsequent section.

IV. 3D Stacked Logic-in-Memory System

In this section, we introduce our overall architecture of the accelerated 3D stacked Logic-in-Memory (LiM) system, and the details of how memory-side accelerators are integrated onto the LiM die. We also present an overview of our proposed software/hardware interface for utilizing the accelerators for the memory-bounded operations.

A. Overall Architecture

Figure 3 shows the overall 3D stacked Logic-in-Memory (LiM) system. The LiM system consists of two major components, i.e., the host multi-core processor for compute-bounded operations and the 3D DRAM stack equipped with accelerators to target the memory-bounded operations. The multi-core processor and the 3D DRAM stack are connected with multiple high-speed serial links, allowing the external bandwidth between them to achieve up to 480 GB/s [11].

In the 3D DRAM stack of LiM system, there exists multiple *DRAM dies*, one *DRAM logic die* and one *LiM die*. The DRAM dies store the data for processing, and the DRAM logic die mainly contains memory controllers for accessing data from the DRAM dies. The DRAM dies and the logic die are connected with high-performance, low-energy, and high bandwidth TSVs (Through Silicon Vias) [12], allowing the internal bandwidth to reach as high as 860 GB/s [13]. We further introduce a LiM die, where multiple accelerators can be integrated to speedup memory-bounded operations, under the DRAM logic die. Both the accelerators on the LiM die and the host processor can leverage the memory controllers on the DRAM logic die to communicate with the DRAM dies.

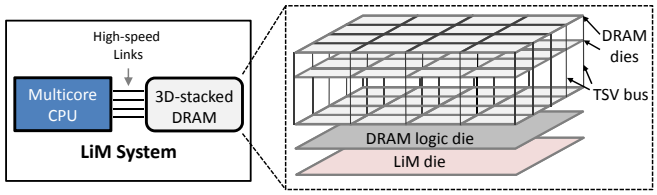

Fig. 3. The overall architecture of 3D stacked Logic-in-Memory system, where high-speed serial links of up to 480 GB/s connects the memory to the host processor, and high bandwidth (860 GB/s) Through Silicon Vias (TSVs) connect different DRAM die stacked on top of each other.

B. LiM Die

On the LiM die, we integrate various accelerators for the aforementioned memory-bounded operations in the STAP algorithm. Accelerators required are `FFT` [5] for the Doppler Processing stage, `Reshape` [14] for data reshape before and after the Doppler Processing stage, and `DOT` [15] and `SCAL` for the normalization and application of the adaptive weights. The basic principle is that each accelerator is mapped to one or more library APIs, such that each library call is equivalent to an accelerator invocation. In addition to the accelerators, a configuration infrastructure is also designed to control and configure the accelerators. The configuration infrastructure consists of the switch interconnect, which connects all the hardware accelerators and the Configuration Unit (CU). The CU parses an *Accelerator Descriptor* that describes the control and configuration of the accelerators, performs the necessary configurations, and invokes the accelerators accordingly.

C. Software/Hardware Interface

We termed the interface between the software and hardware accelerators the *Accelerator Descriptor* (AD). The AD is essentially a physically contiguous memory region, that contains three major parts, i.e., *Command Region* (CR), *Instruction Region* (IR) and *Parameter Region* (PR). The CR is composed of the control command such as `START` and `STOP`, the number of instructions and the starting address of the IR. The IR contains multiple instructions, and each instruction can be either a traditional instruction or a control instruction. The traditional instruction corresponds to one invocation of an accelerator, while the control instruction is related to the control flow operation such as the `LOOP` and `NOP` instruction. For a traditional instruction, in addition to specifying the accelerator to invoke, it also specifies the starting address and the size of accelerator parameters in the PR. Details of the input and output buffers such as *buffer size*, *starting address*, and *stride* for each accelerator invocation are also specified in the PR. The detailed format of the AD is shown in Table II.

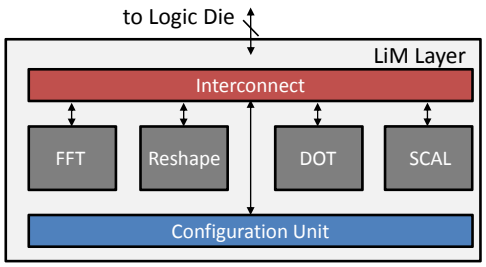

Fig. 4. The architecture of the Logic-in-Memory (LiM) die.

CR				IR				PR							
CTRL CMD	INST NUM	IR ADDR	... ...	INST OP	PARA SIZE	PARA ADDR	... ...	INPUT SIZE	INPUT ADDR	INPUT STRIDE	OUTPUT SIZE	OUTPUT ADDR	OUTPUT STRIDE	PARAMETERS	... ...

TABLE II. THE DETAILED FORMATION OF THE ACCELERATOR DESCRIPTOR (AD).

AD Name	Function	# Accelerators Calls
AD0	Corner turn	1
	Batch FFT	32,768
	Snapshot extraction	1
AD1	Batch inner product	262,144
AD2	Batch inner product	16,777,216
	Batch vector scaling	262,144

TABLE III. MULTIPLE ACCELERATOR INVOCATIONS ARE GROUPED INTO THREE ACCELERATOR DESCRIPTORS (ADS) FOR THE STAP ALGORITHM. NUMBERS INDICATE THE NUMBER OF ACCELERATOR INVOCATION FOR THE LARGE DATA SET.

Offloading of the task from the host to accelerators is performed by transferring the completed AD to the memory space monitored by the Configuration Unit (CU). As there exists non-negligible overheads during task offloading, it is necessary to specify multiple accelerator invocations within one AD so as to reduce the total times of tasks are being offloaded. In the STAP algorithm, where multiple accelerated library calls are within loops, e.g., `DOT` in a series of nested `for` loops, we use the `LOOP` instruction to generate a compact descriptor. For two sequential library calls where the output data of the first library call is exactly the input data of the second library call, these two library calls/accelerator invocations can be placed into the same AD. For the Large data set in PNNL's STAP implementation, a total of 17,334,274 accelerator invocations can be compactly described into only three ADs as shown in Table III.

To generate ADs from the software library calls, we develop a source-to-source compiler to parse the programs instrumented with *directives*. In more detail, the memory-bounded libraries are first instrumented with OpenACC-compatible directives such as `#pragma acc`. Then, our compiler can automatically identify such libraries and translate them to low-level accelerator-related libraries in [16]. At runtime, such low-level libraries can generate corresponding ADs to invoke the accelerators.

V. PERFORMANCE

In this section, we show performance attained by four different implementations, using three different configurations for the third-order Doppler-factored STAP algorithm.

A. Experimental Setup

All experiments were conducted on an Intel Haswell i-4770K multicore processor. To evaluate the performance and energy efficiency of the entire LiM system, we employ the

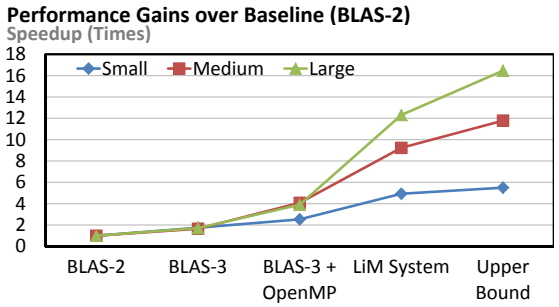

Fig. 5. The performance improvements obtained by using different implementations. The results are normalized to that of the BLAS-2 baseline.

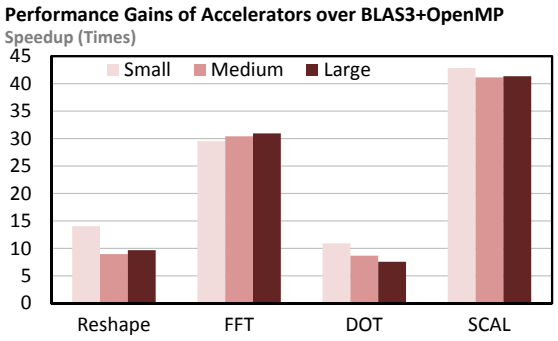

Fig. 6. The performance improvements of different accelerators in the proposed LiM system over the BLAS-3+OpenMP implementation.

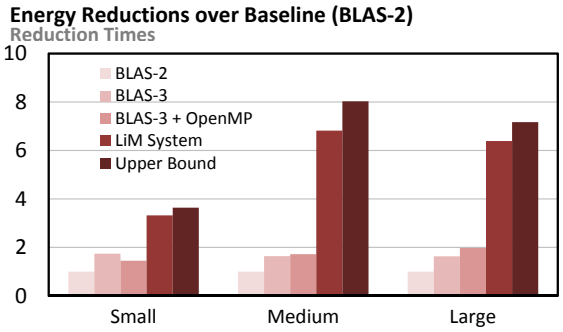

Fig. 7. The energy reduction of different implementations over the `BLAS-2` baseline.

same methodology in [16]. In this methodology, the performance and energy of the host processor are measured by using PAPI (Performance Application Programming Interface) RAPL (Running Average Power Limit) [17] interface, while the performance and energy of the memory-side accelerators are modeled using various simulation and modeling tools, including Synopsis Design Compiler [18], DesignWare, McPAT [19], CACTI-3DD [20], and customized accelerator models. More details of the evaluation methodology can be found in [16].

We ran the three configurations (i.e., Small, Medium, and Large in Table I) using four distinct implementations for all experiments. The baseline is a library-based implementation of PNNL's third-order Doppler-factored STAP algorithm using primarily BLAS 1 and BLAS 2 operations (`BLAS-2`). The second implementation (`BLAS-3`) reduces the number of memory-bounded operations as described in Section III. We show the flexibility of implementing STAP using library and improvement gained by using OpenMP pragmas around the outer-most loops (`BLAS-3 + OpenMP`). This is so that all threads have the largest amount of independent tasks [21]. Finally, the implementation for the LiM system is obtained by compiling the `BLAS-3 + OpenMP` implementation with our experimental compiler (`LiM System`). We also show the theoretical upper bound if the remaining memory-bounded operations require no compute time, and no energy.

B. Performance

In Figure 5, we show the performance comparison of `BLAS-2` baseline, `BLAS-3`, `BLAS-3+OpenMP` and `LiM System`, where the results are normalized to that of the baseline. Compared with the baseline, `BLAS-3`, where memory-bounded operations are converted to compute-bounded operations, improves the performance by 75%, 65%, and 71% for Small, Medium, and Large data set, respectively. By exploiting multithreading, we can achieve 1.53x, 3.08x, and 2.9x performance improvements over the baseline implementation for the Small, Medium, and Large data set, respectively. The performance attained through our LiM system yields the greatest benefit, where more than 11x improvement in performance is attained over the baseline implementation for the large data set.

To demonstrate the benefit of using memory-side accelerators with the 3D stack, we compare the performance of each accelerator and its corresponding software implementation with `BLAS-3+OpenMP`. As shown in Figure 6, accelerators in LiM can significantly and greatly improve the performance. We consistently obtained between five and 40 times speed up in performance for memory-bounded operations over the software implementations.

C. Energy Efficiency

In Figure 7, we compare energy consumption of different implementations, where the results are normalized to that of the baseline (`BLAS-2`) implementation. Converting the memory-bounded operations to compute-bounded operations, we reduce the energy by 42.6%, 39.2%, and 38.7% for the Small, Medium and Large data set, respectively. Exploiting multithreading on the multicore processor (`BLAS-3+OpenMP` implementation), the energy is reduced by 49.6% for the Large data set. An interesting observation is that the energy reduction even decreases for the Small data set, although the execution time is significantly reduced compared with the `BLAS-3` implementation (see Figure 5). The reason is that there is insufficient computation in the small data set to amortize the power consumption resulting from multithreading. Finally, the proposed LiM system can greatly reduce the energy consumption. Compared against the baseline, the LiM implementation reduces energy consumption by a factor of 2.32x, 5.82x, and 5.39x for the Small, Medium and Large data set, respectively, which are 91%, 85%, and 89% of the theoretical upper bound of energy reduction.

In Figure 8, we further compare the energy efficiency in terms of Energy Delay Product (EDP) [22] of different implementations normalized to the EDP of the `BLAS-2` baseline.

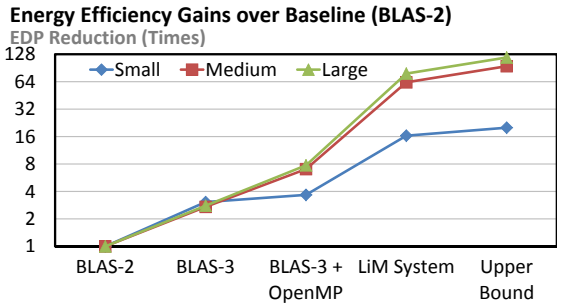

Fig. 8. The energy-delay product (EDP) improvement of different implementations over the `BLAS-2` baseline.

Using the large data set as an example, the `BLAS-3` implementation improves energy efficiency by a factor of 1.79x. Multithreading implementation improves the energy efficiency by a factor of 6.74x. The largest improvement comes from the acceleration with the LiM system, resulting in a total of 77.6x EDP improvement over the baseline. The above results demonstrates the energy efficiency of our proposed software-hardware co-optimized solution.

VI. Conclusion

In this paper, we show that memory-bounded operations take up a significant portion of the compute time for STAP algorithms. As such, we introduced an approach, orthogonal to existing approaches, for optimizing STAP. We reduce and accelerate memory-bounded operations through a reformulation of the STAP algorithm. Acceleration of memory-bounded operations is performed on an accelerated 3D stacked system. These optimization and acceleration results in more than 77.6x and 11.3x improvement in energy efficiency and performance over an basic implementation that uses subroutine calls to optimized libraries (e.g. FFTW and MKL).

While we have used the third-order Doppler-factored STAP algorithm to illustrate our approach, the optimizations we introduced are applicable to many other STAP algorithms as the stages that compute, normalize and apply the adaptive weights using an estimated interference covariance matrix are similar across different algorithms. More importantly, practical applications of STAP typically work with larger array elements, and more potential targets. This larger data set implies that greater energy efficiency gain can be expected since there are more memory-bounded operations that need to be computed.

Acknowledgment

This work was sponsored by the DARPA PERFECT program under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

References

[1] J. Ward, "Space-time adaptive processing for airborne radar," in *Space-Time Adaptive Processing (Ref. No. 1998/241), IEE Colloquium on*, Apr 1998, pp. 2/1–2/6.

[2] M. Wicks, M. Rangaswamy, R. Adve, and T. Hale, "Space-time adaptive processing: a knowledge-based perspective for airborne radar," *IEEE Signal Processing Magazine*, vol. 23, pp. 51–65, 2006.

[3] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[4] Intel, "Math Kernel Library," https://software.intel.com/en-us/intel-mkl, 2015.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "Understanding the design space of dram-optimized hardware FFT accelerators," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 248–255.

[6] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013, http://hpc.pnnl.gov/projects/PERFECT/.

[7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Soft.*, vol. 5, no. 3, pp. 308–323, Sept. 1979.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 14, no. 1, pp. 1–17, March 1988.

[9] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, no. 1, pp. 1–17, March 1990.

[10] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia: SIAM, 1992.

[11] "Hybrid memory cube specification 2.0," 2015. [Online]. Available: http://www.hybridmemorycube.org/specification-v2-download-form/

[12] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2008, pp. 453–464.

[13] B. Akin, J. C. Hoe, and F. Franchetti, "Data reorganization in memory using 3d-stacked dram," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2015.

[14] ——, "Hamlet: Hardware accelerated memory layout transform within 3d-stacked dram," in *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[15] A. Roldao Lopes and G. A. Constantinides, "A fused hybrid floating-point and fixed-point dot-product for fpgas," in *Proceedings of International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, 2010, pp. 157–168.

[16] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3d-stacked memory-side acceleration: Accelerator and system design," in *2nd Workshop on Near Data Processing (WoNDP)*, 2014.

[17] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 189–194.

[18] "Synopsis design compiler." [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler

[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[20] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 33–38.

[21] J. R. Allen and K. Kennedy, "Automatic loop interchange," *SIGPLAN Not.*, vol. 19, no. 6, pp. 233–246, June 1984. [Online]. Available: http://doi.acm.org/10.1145/502949.502897

[22] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, 1996.