GPU-Based Space-Time Adaptive Processing (STAP) for Radar

Thomas M. Benson and Ryan K. Hersey
Sensors and Electromagnetic Applications Laboratory
Georgia Tech Research Institute
Atlanta, Georgia, USA

Edwin Culpepper
Sensors Directorate
Air Force Research Laboratory
Wright-Patterson AFB, OH, USA

***Abstract*—Space-time adaptive processing (STAP) utilizes a two-dimensional adaptive filter to detect targets within a radar data set with speeds similar to the background clutter. While adaptively optimal solutions exist, they are prohibitively computationally intensive. Thus, researchers have developed alternative algorithms with nearly optimal filtering performance and greatly reduced computational intensity. While such alternatives reduce the computational requirements, the computational burden remains significant and efficient implementations of such algorithms remains an area of active research. This paper focuses on an efficient graphics processor unit (GPU) based implementation of the extended factored algorithm (EFA) using the compute unified device architecture (CUDA) framework provided by NVIDIA.**

I. Introduction

Space-time adaptive processing (STAP) utilizes a two-dimensional adaptive filter to detect targets within a radar data set with speeds similar to the background clutter [1]–[5]. The direct joint-domain STAP implementation applies the adaptive filter jointly in the spatial (channel) and slow-time (pulse) domains. While adaptively optimal, the joint-domain implementation requires more training data than is practically available and is extremely computationally intensive [2]. Alternatively, rank-reduced post-Doppler STAP implementations provide nearly optimal performance while significantly reducing training requirements and the overall computational burden. Specific post-Doppler STAP algorithms include the extended factored algorithm (EFA) and the joint-domain localized (JDL) algorithm [6], [7]. This paper focuses on implementations of EFA for an airborne radar system.

Incoming data rates for radar platforms can be quite large, and EFA exhibits substantial computational complexity, so computational efficiency is an important consideration for STAP processing. Furthermore, airborne platforms tend to have size, weight, and power (SWaP) constraints as well as ruggedization requirements (shock, vibration, etc.) that both restrict the available hardware that can be effectively deployed in such environments as well as the relative importance of computational efficiency (or, relatedly, power efficiency) due to the non-viability of adding additional hardware.

In this work, we consider the implementation of EFA on NVIDIA graphics processing units (GPUs) using the compute unified device architecture (CUDA) framework. GPUs offer a compelling platform for the implementation of sophisticated sensor processing techniques due to their availability in ruggedized form factors from multiple vendors and their relatively high peak-performance to power ratios, as measured by gigaflops per Watt (GFLOPS/W). Furthermore, modern GPUs offer very high memory bandwidth for certain memory access patterns, which is critical for achieving near-peak performance on many applications. In terms of programmability, GPU software development is typically more complex than traditional CPU software development, but substantially less complex than hardware description language (HDL) implementations for field programmable gate arrays (FPGAs), especially for complicated algorithms. On the other hand, FPGAs offer several advantages, including potentially lower power consumption than GPUs for equivalent processing and generally highly predictable runtimes. Thus, GPUs present a middle ground between CPUs and FPGAs from the perspective of programmability and performance per Watt [8].

II. Extended Factored Algorithm (EFA)

We employ the extended factored algorithm (EFA) introduced by DiPietro [6] with a block training approach to estimate the required covariance matrices. We introduce EFA briefly in this section and proceed to describe each of the steps in greater detail in the following sections. Consider a radar data set $\mathbf{x}$ consisting of M array elements or spatial channels, L range bins per pulse, and N_{CPI} pulses within a coherent processing interval (CPI). Thus, from a traditional radar nomenclature perspective, L and N_{CPI} correspond to the fast-time and slow-time dimensions, respectively, and N_{CPI} and M correspond to the temporal and spatial domains, respectively. Furthermore, consider a data set $\mathbf{X}$ where we have transformed the pulses to Doppler space by applying a discrete Fourier transform of length $N_D \geq N_{CPI}$ along the pulse dimension, potentially after having applied a window function.

Assume P steering vectors of interest with p-th steering vector $\mathbf{v}(p) \in \mathbb{C}^{M \times 1}$. From the perspective of an EFA implementation, steering vectors are provided as input that will ultimately provide the right-hand sides of linear systems to be solved. The output datacube, $\mathbf{y}$, generated by EFA will have dimensions $P \times L \times N_D$. Output value $\mathbf{y}(p, b, n)$ is given by applying an adaptive weighting vector to an associated portion of $\mathbf{X}$ where the weighting vector is computed by solving a linear system involving the estimated covariance for range bin b and Doppler bin n. In practice, the covariance will be approximated as a sum of outer products, $\hat{\mathbf{R}}(b, n) = \sum_{j \in J(b,n)} \mathbf{X}_j \mathbf{X}_j^H$, where $\mathbf{X}_j$ is a to-be-defined subset of $\mathbf{X}$ in vector form and $J(k, n)$ is a training set.

For EFA, the temporal degrees of freedom for Doppler bin n will include the immediate Doppler neighborhood of n. Using the convention from [6], order two processing corresponds to considering three Doppler bins centered on n (i.e., $n-1, n$, and $n+1$). This approach yields three temporal degrees of freedom, which we indicate with the variable T_{DOF}. While other values are possible, we consider the $T_{DOF}=3$ case throughout this paper for concreteness. Furthermore, we apply block training in range. Thus, we subdivide the L range bins into blocks with B range bins per block. For range bins in a given block L_b, we acquire training data from the two adjacent blocks (i.e., L_{b-1} and L_{b+1}). Edge cases exist for the Doppler and training neighborhoods and are addressed in Section III-B.

III. Computational Analysis

Prior to considering the GPU implementation, we will first derive the computational requirements for implementing EFA by analyzing the following steps independently: (1) Doppler processing, (2) covariance estimation, (3) linear system solver, and (4) adaptive matched filter (AMF) weighting. As input to EFA, we have the data cube $\mathbf{x} \in \mathbb{C}^{M \times L \times N_{CPI}}$ and P steering vectors with $\mathbf{v}(p) \in \mathbb{C}^{M \times 1}$. The output datacube will be given by $\mathbf{y} \in \mathbb{R}^{P \times L \times N_D}$. While complex output is also possible, we consider the case of converting to the power domain after applying the adaptive filters.

A. Doppler Processing

Doppler processing maps a data cube $\mathbf{x} \in \mathbb{C}^{M \times L \times N_{CPI}}$ to $\mathbf{X} \in \mathbb{C}^{M \times L \times N_D}$ through the independent application of a length $N_D \geq N_{CPI}$ discrete Fourier transform (DFT), potentially with a Doppler windowing function $w_D \in \mathbb{R}^{N_{CPI}}$, to each pairing of channel and range bin. The windowing function is real-valued and corresponds to an element-wise multiplication in the pulse dimension. The DFT will presumably be implemented using the fast Fourier transform (FFT) and N_D can be chosen such that the FFT is efficient (e.g., by setting N_D to be a power of two). Depending on the memory layout of the incoming data cube, it may be more efficient to reorganize the data cube such that the N_{CPI} samples for a given channel and range bin are stored contiguously and thus the FFT can be applied to a contiguous array. Assuming that a window is used, the Doppler processing step requires roughly $\mathcal{O}(M \cdot L \cdot N_D \cdot \log_2 N_D)$ floating point operations (FLOPs).

B. Covariance Estimation

In general, with T_{DOF} temporal degrees of freedom, the covariance estimation generates a set of $L \times N_D$ matrices of dimension $(M \cdot T_{DOF}) \times (M \cdot T_{DOF})$. However, we employ block training such that a single covariance matrix will be associated with multiple range bins in a range block. With block factor B, we have $L_B = \lceil L/B \rceil$ range blocks where $\lceil x \rceil$ represents the smallest integer not smaller than x. For simplicity, we assume that B divides evenly into L. Thus, covariance estimation with block training generates a set of $L_B \times N_D$ covariance matrices, which reduces the computational burden of the subsequent linear system solver by a factor of B.

For notational purposes, we use $L_b \in \{0, \ldots, L_B - 1\}$ and $b \in \{0, \ldots, L-1\}$ to represent indices corresponding to a range block and a range bin, respectively. For each range block L_b, we form the matrix $\mathbf{Q}(L_b, n) \in \mathbb{C}^{(M \cdot T_{DOF}) \times B}$ with rows corresponding to Doppler samples within the temporal window for Doppler bin n and all channels thereof and columns corresponding to the B range bins in block L_b. Per-block covariance estimates, $\mathbf{T}(L_b, n)$, can then be computed via outer products, i.e.,

$$\mathbf{T}(L_b, n) = \frac{\mathbf{Q}(L_b, n)\mathbf{Q}(L_b, n)^H}{B}.$$

Finally, for a given range block, the covariance estimate $\hat{\mathbf{R}}(L_b, n)$ is given by a combination of neighborhood training data. The number of neighboring training blocks can be adjusted and certain blocks (i.e., guard blocks) can be excluded, but for simplicity we consider the mean of the adjacent training blocks. Therefore, the covariance estimate for range block L_b and Doppler bin n is

$$\hat{\mathbf{R}}(L_b, n) = \frac{\mathbf{T}(L_b - 1, n) + \mathbf{T}(L_b + 1, n)}{2}.$$

There are boundary conditions that must be handled for the edge blocks when computing $\hat{\mathbf{R}}$ and the minimum and maximum Doppler values when defining $\mathbf{Q}$. The former case is handled using the two nearest range blocks to generate $\hat{\mathbf{R}}$ (e.g., $\mathbf{T}(1, n)$ and $\mathbf{T}(2, n)$ contribute to $\hat{\mathbf{R}}(0, n)$) and the latter case is handled by wrapping in Doppler (i.e., -1 wraps to $N_D - 1$ and N_D wraps to 0). Both of these boundary approaches generalize to wider temporal windows and alternative training strategies.

Furthermore, both the $\mathbf{T}$ and $\hat{\mathbf{R}}$ matrices are conjugate symmetric due to the symmetry associated with the outer product. There are additionally some redundancies among the $\mathbf{T}$ matrices that can potentially be exploited to reduce the computational load. We consider those redundancies when discussing the GPU implementation. The overall computational complexity ignoring such symmetries is given by $\mathcal{O}(L \cdot N_D \cdot (M \cdot T_{DOF})^2)$.

C. Linear System Solver

There are many algorithmic options for solving the linear systems $\hat{\mathbf{R}}(L_b, n)\mathbf{w}(L_b, n) = \mathbf{v}(p)$. The covariance estimation matrices are conjugate symmetric and positive semidefinite by construction. Furthermore, given sufficient training and linear independence of training data due to the presence of noise, the covariance estimation matrices are typically positive definite [1]. Positive definiteness enables additional algorithms, such as Cholesky factorization or Gaussian (Gauss-Jordan) elimination with no pivoting requirement. It is not necessary to form the explicit inverse matrices $\hat{\mathbf{R}}^{-1}(L_b, n)$ in order to solve the linear systems.

For typical parameter sets, the linear system solver step will be applied to many small linear systems. In particular, there are $L_B \times N_D$ systems with matrices of dimension $M \cdot T_{DOF} \times M \cdot T_{DOF}$ and P right-hand sides. We explored three options for solving these systems: (1) utilizing existing GPU-based linear algebra libraries, (2) applying Cholesky factorization paired with forward and back substitution, and (3) applying Gauss-Jordan elimination to an augmented matrix form.

While many high-quality linear algebra libraries exist, including cuBLAS for NVIDIA GPUs, these libraries typically focus on solving large linear systems rather than a large number of small linear systems. However, the so-called batch mode operations that solve many linear systems are increasingly being incorporated into such libraries. As of this writing, cuBLAS includes a batched LU decomposition routine as well as a batched triangular system solver. However, our implementation currently achieves higher performance than that achieved by utilizing the library, partially because we compose additional operations into our kernels. For example, the combination of neighboring training blocks to form the final covariance matrices described in Section III-B is performed in the same kernel that factorizes or solves the system. Thus, in a custom implementation, the final covariance matrices only exist in GPU shared memory, which corresponds to a single read from global memory, whereas utilizing a library requires an additional read-write cycle to form the input to the library call. However, as libraries evolve to be more comprehensive, shifting to library based linear solvers may be ideal, especially given that libraries typically evolve to exploit new hardware and would thus partially future-proof the implementation.

The Cholesky-based solution involves factorizing the covariance matrices $\hat{\mathbf{R}}(L_b, n)$ into $\mathbf{L}(L_b, n)\mathbf{L}(L_b, n)^H$ and then applying forward and back substitution as follows:

(1) Solve $\mathbf{L}(L_b, n)\mathbf{z}(p, L_b, n) = \mathbf{v}(p)$ (forward sub)

(2) Solve $\mathbf{L}(L_b, n)^H\mathbf{w}(p, L_b, n) = \mathbf{z}(p, L_b, n)$ (back sub).

Alternatively, we can form the $M \cdot T_{DOF} \times (M \cdot T_{DOF} + P)$ augmented matrix $\left[\hat{\mathbf{R}}(L_b, n) \mid \mathbf{v}\right]$ and apply Gauss-Jordan elimination to the resulting augmented system matrix. After applying elimination, the augmented matrix will have the form $[\mathbf{I} \mid \mathbf{w}(L_b, n)]$ where $\mathbf{I}$ is the appropriately sized identity matrix. As a consequence of $\mathbf{w}(L_b, n)$ being produced directly by the Gauss-Jordan elimination process, no additional kernels are required for that case whereas natural implementations of the Cholesky factorization case require an additional write and read of the intermediate $\mathbf{L}$ matrices.

In terms of computational complexity, both Cholesky factorization and Gauss-Jordan elimination have complexity $\mathcal{O}((M \cdot T_{DOF})^3)$. Gauss-Jordan elimination carries a higher constant of proportionality and is thus less FLOP efficient. However, because there are P right-hand sides per linear system corresponding to the various steering vectors, the system solve portions will have computational complexity $\mathcal{O}(P \cdot (M \cdot T_{DOF})^2)$ with a lower constant of proportionality for the Gauss-Jordan solver. Thus, the most FLOP efficient approach overall depends on the specific values of M, T_{DOF}, and P. Because we solve for $L_B \times N_D$ such systems, the computational complexity of the Cholesky and Gauss-Jordan approaches are given by $\mathcal{O}(L_B \cdot N_D \cdot ((M \cdot T_{DOF})^3 + P \cdot (M \cdot T_{DOF})^2))$. In addition to FLOP efficiency, these approaches correspond to substantially different memory access patterns, which can in practice be more important than modest differences in required operation counts.

D. AMF Weighting

The power domain output $\mathbf{y} \in \mathbb{R}^{P \times L \times N_D}$ is computed by applying the adaptive weights to the appropriate elements from the Doppler processed data cube. Let $\mathcal{S}(\mathbf{X}, b, n)$ be an operator that extracts as a column vector the space-Doppler window corresponding to range bin b and Doppler bin n from data cube $\mathbf{X}$ (i.e., $\mathcal{S}$ corresponds to a column from the $\mathbf{Q}$ matrices in Section III-B). Output element $\mathbf{y}(p, b, n)$ is then given by

$$\mathbf{y}(p, b, n) = \beta(p, L_b, n) \left\| \mathbf{w}(p, L_b, n)^H \mathcal{S}(\mathbf{X}, b, n) \right\|^2$$

where

$$\beta(p, L_b, n) = \frac{1}{\sqrt{\mathbf{w}(p, L_b, n)^H \mathbf{v}(p)}}.$$

Therefore, applying the adaptive weighting involves $P \times (L + L_B) \times N_D$ inner products of length $M \cdot T_{DOF}$, plus the squaring and square root operations involved in the power and normalization calculations, yielding a computational complexity of $\mathcal{O}(P \cdot L \cdot N_D \cdot (M \cdot T_{DOF}))$.

IV. GPU Implementation Details

We present a very brief introduction to modern Fermi and Kepler-branded NVIDIA GPU architectures and the associated CUDA programming model. Additional detail is available in the CUDA Programming Guide [9] as well as many other sources. A modern NVIDIA GPU consists of several symmetric multiprocessors (denoted SM for Fermi and SMX for Kepler). Fermi-generation GPUs feature 32 cores per SM whereas Kepler-generation GPUs feature 192 cores per SMX. In total, the GPU has hundreds to thousands of cores that can operate simultaneously. A block of code executed on the GPU is typically known as a kernel and is written from the perspective of a single thread with a parameterized thread and block index. Blocks of threads can be executed on an SM with multiple blocks used per SM, up to various hardware limits, in order to minimize memory access latency. Blocks are oriented into grids and the CUDA driver manages mapping blocks to specific multiprocessors during run-time. In order to extract parallelism from a problem, the workload needs to be mapped to the grids and blocks supported by CUDA.

The memory hierarchy includes multiple cache levels, shared memory (i.e., fast user-managed memory), constant memory, texture memory, and the relatively slower global memory. Fermi and Kepler both include up to 48 KiB of shared memory per SM[*] that can be used by blocks running on the SM in order to reduce accesses to the relatively slower global memory. Both GPU generations include modest-sized L2 caches of 768 KiB for Fermi and 1536 KiB for Kepler.

Efficient GPU implementations require the exploitation of massive parallelism in combination with a careful consideration of interactions with the memory hierarchy. In particular, when possible, low-level shared memory or memory caches should be utilized in order to minimize latency incurred by global memory accesses. When global memory access is required, care should be taken to access memory in a coalesced fashion when possible. Coalesced memory accesses apply to memory access patterns meeting certain requirements, such

[*]Both GPU generations feature 64 KiB of memory per SM or SMX with user-configurable combinations of shared memory and device-managed memory, which can either operate as an L1 cache in the case of Fermi or a spilled-register cache in the case of Kepler.

TABLE I. Data cube dimensions and EFA parameters used for benchmarking.

Parameter	Variable	Value
Spatial channels/elements	M	4
Pulses per CPI	N_{CPI}	128
Doppler bins	N_D	256
Range bins	L	512
Range bins per training block	B	32
Temporal degrees of freedom	T_{DOF}	3
Steering vectors	P	32

as each thread t simultaneously referencing array location `array[t]`, so that the memory accesses are contiguous and can thus exploit the wide data paths to global memory.

While describing our implementation, we will consider the steps outlined in Section III independently for simplicity, although it would also be possible for an implementation to compose the various steps (e.g., by applying multiple steps within a single kernel). The utilization of shared memory by CUDA kernels ultimately impacts occupancy and final performance and thus we consider the specific collection of input dimensions and parameters shown in Table I when presenting the following implementation details. For different parameter sets, the preferred implementations may change, although we have tried several variations with no substantial change in the conclusions.

A. Doppler Processing Implementation

The Doppler processing step is fairly straightforward due to the availability of high-quality FFT kernels provided by the CUFFT [10] library. The CUFFT library requires that the data to-be-transformed be stored contiguously in memory. In the typical case that $N_D > N_{CPI}$, zero padding will be required prior to initiating the FFTs. Furthermore, the incoming datacube may be stored such that slow-time samples are not stored contiguously. For our implementation, we assume that the data cube is stored such that the complex sample for channel m, range bin b, and pulse n has index $(m \cdot N_{CPI} + n) \cdot L + b$. Thus, there is a kernel that implements a corner turn, applies the windowing, and adds the zero padding prior to invoking CUFFT. After Doppler processing, the data cube is stored such that the complex sample for channel m, range bin b, and Doppler bin n has index $(m \cdot L + b) \cdot N_D + n$. Finally, a kernel reorganizes the data in a manner similar to the MATLAB `fftshift` function in order to shift the DC frequency to the central Doppler bin.

B. Covariance Estimation Implementation

As noted in Section III-B, the covariance matrices are symmetric and thus only the upper or lower diagonal elements need to be computed. However, there are additional redundancies that can be exploited to further reduce the computational load. Consider the individual entries of the matrices $\mathbf{T}(b, n)$ and $\mathbf{T}(b, n+1)$ for $n < N_D - 1$. Each matrix entry corresponds to an inner product of a pair of rows from the associated $\mathbf{Q}(b, n)$ and $\mathbf{Q}(b, n+1)$ matrices. Furthermore, the rows from $\mathbf{Q}$ represent a unique pairing of channel and temporal degree of freedom for the associated Doppler bin. However, because the temporal windows overlap from one Doppler bin to the next, the overlapping portion in Doppler corresponds to redundant entries in $\mathbf{T}(b, n)$ and $\mathbf{T}(b, n+1)$. Furthermore, there are redundancies between $\mathbf{T}(b, n+1)$ and $\mathbf{T}(b, n+2)$ such that only one in T_{DOF} of the entries in a given $\mathbf{T}$ matrix are unique. Therefore, while the covariance estimation output conceptually consists of $L_B \times N_D \times (M \cdot T_{DOF})^2$ inner products of length B, we only need to compute the inner products for unique entries. Note that while redundancy reduction is an effective optimization in this case, certain algorithmic changes, such as incorporating power comparable training (PCT) [11], can eliminate the redundancies in the sliding temporal window.

Therefore, the implementation for the covariance training matrices is split into two kernels: the first generates unique inner products (up to conjugate symmetry) that are needed by at least one entry in the training matrices and the second rearranges the results of the first kernel into matrices with redundant values stored in each matrix for which they are needed. In particular, the second kernel involves indexing and memory access, but the only floating point operations correspond to conjugating entries in the case that the first kernel calculated the conjugate of the value needed for a particular matrix entry. The first kernel utilizes thread blocks with N_D threads per block where each thread computes an inner product over B values. Because there is little reuse of data within a thread block, no shared memory is used for the covariance estimation implementation.

C. Linear System Solver Implementation

For the nominal case of Hermitian positive definite covariance matrices, matrix decompositions can be implemented using any of QR decomposition, LU decomposition, or Cholesky decomposition. Furthermore, Gauss-Jordan elimination can be applied to the augmented system matrix presented in Section III-C without a pivoting requirement. While we implemented both the Cholesky and Gauss-Jordan based approaches, Gauss-Jordan elimination proved more efficient for our particular parameter set and thus the following description focuses on the Gauss-Jordan case. In addition, we explored utilizing batched matrix decompositions from existing libraries (e.g., batched LU decomposition from cuBLAS), but we achieved higher performance with custom kernels, partially because we compose additional operations into the kernel as will be described shortly.

The covariance matrices have dimension $(M \cdot T_{DOF}) \times (M \cdot T_{DOF})$, or 12×12 for our benchmark parameter set. Thus, with single precision complex floating point entries, each covariance matrix requires 1152 bytes of storage. Similarly, the P steering vectors, each with $M \cdot T_{DOF}$ entries, require 3072 bytes of storage for a total footprint of 4224 bytes. Because most of these entries will be accessed repeatedly, they are stored in shared memory with one thread block managing a single covariance matrix and set of steering vectors.

Furthermore, the covariance matrix for a given range block is the average of the training matrices from the adjacent blocks, so that the required memory accesses and averaging are performed as part of the Gauss-Jordan and Cholesky kernels with the resulting covariance matrix being written directly into shared memory. Therefore, the final covariance matrices are never written to global GPU memory, whereas utilizing an existing library would require an intermediate kernel to compute the covariance matrices and write the result to global memory as input for the library call. Finally, although the covariance matrix is conjugate symmetric and thus only a portion of the matrix needs to be stored, we allocate the full 4224 bytes of shared memory to support natural indexing within the algorithm implementations.

Thread blocks consist of $(M \cdot T_{DOF})/B_T \times (T_{DOF} + P)$ threads, where B_T is a thread blocking factor to be described shortly, and each thread block performs Gauss-Jordan elimination on a single augmented matrix. The load balancing in this case is not ideal. While the threads managing the steering vectors have operations to apply at each step, the threads managing the covariance matrix have uneven workloads – in fact, the threads for the upper triangular portion of the covariance matrix are not necessary due to conjugate symmetry. Therefore, this thread allocation strategy works best in cases where the number of output beams is large relative to the number of channels and temporal degrees of freedom. Similar asymmetric thread workloads exist for Cholesky factorization as well, at least for natural thread mappings.

The thread blocking strategy utilizes a single thread to manage B_T rows of data, which increases opportunities for instruction level parallelism, improves load balancing, and increases the effective use of available registers. Occupancy corresponds roughly to the utilization efficiency of the available GPU resources on a given symmetric multiprocessor (SM) and can be limited by register utilization, total number of blocks, total number of threads, or shared memory utilization. In our case, if $B_T = 1$, then occupancy is always limited by the total number of available threads on a given SM (i.e., 1536 and 2048 for Fermi and Kepler generation NVIDIA hardware, respectively). As B_T increases, more blocks can be assigned per SM, which requires storing more augmented matrices and thus shared memory can become a bottleneck. In practice, we choose B_T empirically based on achieved performance with $B_T = 3$ for our current implementation.

D. AMF Weighting Implementation

An adaptive weighting vector for a given steering vector and Doppler bin is invariant within its range block and thus the weights can be stored in shared memory and applied to an entire range block. Therefore, our thread grid has dimension $N_D \times L_B$ with B threads per block and each thread in turn applies the adaptive weighting vectors for all beams of a given range bin. This strategy requires storing $(M \cdot T_{DOF}) \times P$ complex values per block (3072 bytes in our case) in shared memory. Furthermore, each thread stores $M \cdot T_{DOF}$ elements from $\mathbf{X}$ locally in registers and applies the P adaptive weighting vectors to this local vector to generate P output elements of $\mathbf{y}$. While this approach features few threads per block, the use of registers and shared memory minimizes the need for a large number of threads per SM, which primarily serve to hide global memory access latency.

V. Results

We present performance results using the parameters given in Table I on NVIDIA Tesla M2090, Tesla K20c, and Quadro 3000M GPUs. Several relevant performance metrics are given in Table II for each of the GPUs. For the M2090 and K20c

TABLE II. Performance specifications for measured GPUs. All reported values are theoretical peaks. GFLOPS corresponds to single precision gigaflops per second and the thermal design power (TDP) is used as a surrogate for power draw.

GPU Model	GFLOPS	Memory BW	TDP	GFLOPS/Watt
M2090	1331	177 GB/s	250 W	5.32
K20c	3519	208 GB/s	225 W	15.64
Q3000M	432	80 GB/s	75 W	5.76

TABLE III. Average runtimes (in milliseconds) for each stage of EFA on all target GPUs.

	M2090	K20c	Q3000M
Doppler Processing	0.30 ms	0.24 ms	0.80 ms
Covariance Estimation	0.82 ms	0.52 ms	2.24 ms
Linear System Solves	1.21 ms	0.88 ms	5.20 ms
Adaptive Weighting	1.75 ms	1.31 ms	7.69 ms
Total	4.07 ms	2.95 ms	15.93 ms

results, error correction (ECC) is enabled for memory, which decreases the available bandwidth. These three platforms demonstrate the differences between the Fermi generation (M2090, Q3000M) and Kepler generation (K20c) GPUs as well as the differences between server-class hardware with high power utilization (M2090 and K20c) and mobile hardware with more modest power requirements (Q3000M). Furthermore, the Q3000M and M2090 are different variations of the Fermi architecture, offering compute capabilities of 2.1 and 2.0, respectively. The Q3000M is available in an embedded form factor from several vendors with support for varying levels of ruggedization.

All of the following results were obtained using CUDA 5.0 with driver version 310.44 and code generated according to the highest supported compute capability (i.e., 2.0, 2.1, and 3.5 for the M2090, Q3000M, and K20c, respectively). The reported timings correspond to the average values obtained by processing 32 data sets. Although the same CUDA code is compiled and used for testing on all platforms, the code was originally developed and tuned for performance on the M2090 and thus may be slightly performance-biased toward that platform.

The execution times associated with the various processing stages for all evaluated GPUs are shown in Table III. Figure 1

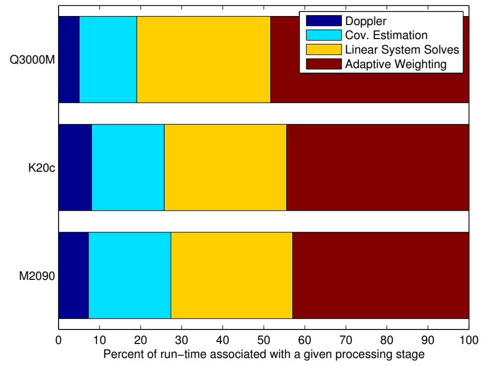

Fig. 1. Breakdown of relative execution time among EFA stages on each of the evaluated GPUs.

depicts the same data in normalized bar chart form. Several observations can be made from the resulting data. Firstly, although the linear system solves may be expected to consume the most computational time due to their cubic scaling, the cubic growth is in terms of $M \cdot T_{DOF}$, which for this case is relatively modest. In particular, $M \cdot T_{DOF} < B$, so adaptive weighting features a higher workload than the system solver. Moreover, the RMB rule, named after the authors who observed it, implies that training over approximately $2 \cdot M \cdot T_{DOF}$ samples corresponds to a roughly 3 dB performance loss under certain assumptions [12]. Thus, we would expect the inequality $M \cdot T_{DOF} < B$ to hold for other parameter sets in order to prevent performance loss.

Secondly, the Q3000M results in several cases seem to be inconsistent with the specifications shown in Table III. Relative to the M2090, the Q3000M has approximately one-third and one-half of the peak GFLOPS and memory bandwidth values, respectively. While Doppler processing and covariance estimation fall within the performance range implied by the specification differences, both the linear system solver and adaptive weighting steps are more than four times slower on the Q3000M than the M2090. We have not definitively determined the cause of this discrepancy, but it is notable that the Q3000M features 48 cores per SM with some scheduler differences and thus will interact with block sizes and instruction-level parallelism differently than the M2090. Finally, although the K20c offers approximately three times more peak theoretical GFLOPS than the M2090, the observed performance increase for these kernels ranges from 1.3x to 1.5x. As noted above, the thread block sizes and blocking factors have been tuned to optimize performance on the M2090, but have not yet been retuned specifically for the K20c and Q3000M. However, memory performance is a major factor in the K20c results. Although the K20c triples the peak theoretical GFLOPS performance relative to the M2090, the memory bandwidth increases by a modest 18%. Thus, memory bound kernels can be expected to achieve improvements more consistent with the relative memory bandwidth rather than the relative peak floating point performance.

We now shift to considering the efficiency of the implementations on the various platforms in terms of performance per Watt. While we do not have exact floating point operation counts for all of the kernels from which to calculate the percentage-of-peak performance, we can instead calculate the data sets per second that each platform can process and use the thermal design power (TDP) as a surrogate for power consumption from which to compute data sets processed per second per Watt. While the power draw for a given device may be substantially below TDP[†] (e.g., in the case that the implementation does not achieve near-peak utilization), our metric allows us to roughly compare the available platforms. Using this approach, we obtain data sets per second per Watt values of 0.98 for the M2090, 1.51 for the K20c, and 0.84 for the Q3000M. Thus, the M2090 and Q3000M offer similar performance per Watt, especially given that we expect to improve performance on the Q3000M via kernel parameter tuning. The Kepler-generation K20c offers substantially improved performance per Watt relative to the previous generation Fermi hardware, which highlights a major advantage of utilizing commodity hardware with rapid upgrade cycles.

[†]The TDP does not correspond to maximum power draw, but rather to the amount of power the cooling system should be able to dissipate for nominal applications. Thus, it is possible that power draw could exceed TDP.

VI. Conclusions

We have demonstrated that high-performance GPU-based STAP implementations for radar processing are feasible. These implementations can be deployed on multiple devices with varying overall performance characteristics, power requirements, and ruggedization capabilities. Furthermore, the newer generation Kepler NVIDIA GPUs were shown to offer substantially improved performance per Watt relative to the previous Fermi generation with no source code modifications required.

Many of the implementations for specific processing stages depend heavily upon utilizing shared memory in order to optimize performance. As a result, performance is rather sensitive to changes in parameters, especially the number of channels, output beams, and temporal degrees of freedom. In some cases, implementations would likely change for different parameter sets, although we have tested variations in output beam counts with no substantive changes in the relative performance conclusions. Modifications motivated by parameter changes could range from simply changing a blocking factor to redesigning the entire shared memory utilization strategy. Considering optimized implementations for a wider range of both parameter sets and hardware devices, especially if such implementations can be generated automatically, represents a fruitful area for further research.

References

[1] W. L. Melvin and J. A. Scheer, Eds., *Principles of Modern Radar: Advanced Techniques*. SciTech Publishing, 2013, ch. 10.

[2] W. Melvin, "A STAP overview," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 19, no. 1, pp. 19–35, 2004.

[3] J. R. Guerci, *Space-Time Adaptive Processing for Radar*. Norwood, MA: Artech House, 2003.

[4] R. Klemm, *Principles of Space-time Adaptive Processing*, ser. IEE Radar, Sonar, Navigation and Avionics Series, 12, I. of Electrical Engineers, Ed. Institution for Engineering and Technology, 2002.

[5] ——, *Space-Time Adaptive Processing: Principles and Applications*, ser. IEE Radar, Sonar, Navigation and Avionics 9, I. of Electrical Engineers, Ed. Institution for Engineering and Technology, 1998.

[6] R. DiPietro, "Extended factored space-time processing for airborne radar systems," in *Signals, Systems and Computers, 1992. 1992 Conference Record of The Twenty-Sixth Asilomar Conference on*, vol. 1, 1992, pp. 425–430.

[7] H. Wang and L. Cai, "On adaptive spatial-temporal processing for airborne surveillance radar systems," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 660–670, 1994.

[8] M. Bales, T. Benson, R. Dickerson, D. Campbell, R. Hersey, and E. Culpepper, "Real-time implementations of ordered-statistic CFAR," in *Radar Conference (RADAR), 2012 IEEE*, 2012, pp. 896–901.

[9] NVIDIA, "NVIDIA CUDA C programming guide: Version 5.0," NVIDIA Corporation, Tech. Rep., 2012, available as of this writing at http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[10] ——, "CUFFT library," NVIDIA Corporation, Tech. Rep., 2012, available as of this writing at http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf.

[11] T. Selee, K. Bing, and W. Melvin, "STAP application in mountainous terrain: Challenges and strategies," in *Radar Conference (RADAR), 2012 IEEE*, 2012, pp. 824–829.

[12] I. S. Reed, J. D. Mallett, and L. E. Brennan, "Rapid convergence rate in adaptive arrays," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. AES-10, no. 6, pp. 853–863, 1974.