FFT-Based Spatio-Temporal Noise Covariance Matrix Inversion on Hybrid Multicore Processor Systems

Jacob Barhen*   Travis Humble   Pramita Mitra*
Charlotte Kotas   Neena Imam   Bryan Schleck

Abstract. The emergence of streaming multicore processors with multi-SIMD architectures opens unprecedented opportunities for executing many sophisticated signal processing algorithms faster and with a much lower energy budget. Here we report on the development, implementation, and demonstration of a novel, massively parallel computational scheme for inverting the spatio-temporal covariance matrix associated with ambient noise in signal detection algorithms. Our methodology involves extensive use of the FFT, for which we exploit the capabilities of leading hybrid multicore processors, including the IBM Cell, the Nvidia Tesla, and the Coherent Logix HyperX.

Index Terms — FFT, multicore processors, Cell, Tesla, HyperX, spatio-temporal data whitening

I. Introduction

As stealthier underwater targets become more pervasive, there is a need to deploy ever more performing sensor arrays. To fully exploit the information contained in data measured by these novel devices often requires the use of unconventional algorithms that exhibit growing computational complexity as function of the number of acoustic channels and the number of complex samples in each observation window.

For example, signal whitening has long been recognized as an essential stage in processing sonar array data [1], and the Fast Fourier Transform (FFT) has played, for many years, a fundamental role in that context [2-4]. The unconventional twice whitening paradigm includes the inversion of the spatio-temporal covariance matrix for ambient noise. The computational challenge one then faces stems from the following considerations.

II. Computational Platforms

Four parameters are of paramount importance when evaluating the relevance of emerging computational platforms for time-critical, embedded applications. They are: computational speed, communication speed (the I/O and inter-core data transfer rates), the power dissipated (usually measured in pico-Joules per floating point operation), and the processor footprint. For each of these parameters, one can compare the performance of an algorithm for different
hardware platforms and software (e.g., compilers) tools. In that context, several multicore processors are of relevance for computationally intensive maritime sensing applications.

II.1 The HyperX hx3100 Processor

The hx3100 processor is the latest entry in the HyperX family of ultra low power, massively parallel processors produced by Coherent Logix, Inc [8]. Because of its ultra-low power consumption, it is a very strong contender not only for maritime sensing computations, but also for those applications that can substantially benefit from an MIMD capability in conjunction with real-time reconfigurability.

a. HyperX Architectural Overview

The hx3100 processor consists of an array of processing elements (PEs) of size 10-by-10, embedded on an 11-by-11 array of data management and routing units (DMRs). It is illustrated in Figure 1. At a system clock frequency of 500 MHz, the maximum chip-level throughput for 32-bit floating-point operations is 25 GFLOPS. Alternatively, when power consumption is prioritized, performance can be measured for the current release as 16 GFLOPS/Watt. This translates into energy consumption on the order of 10 pico-Joules (pJ) per instruction, which rivals the performance of dedicated ASIC designs.

The DMR network provides Direct Memory Access (DMA) for the PEs to both on-chip and off-chip memory. Each DMR has 8 KB of SRAM, and operates at a read/write cycle rate of 500 MHz, while the eight independent DMA engines within a DMR may act in parallel. A PE directly accesses data memory in four neighbouring DMRs, such that 32 KB of data is addressable by any given PE. In addition, when a subset of PEs shares direct access to the same DMR, the associated memory space may act as shared memory between PEs. This inter-connectedness provides for a mixture of shared and distributed memory hierarchies. Each DMR consists of 8 16-bit DMA engines that route memory requests from neighbouring PEs and support routing memory requests managed by other DMRs. In addition to supporting on-chip DMAs, the DMRs handle requests to off-chip memory, including the eight DDR2 DRAM ports. Moreover, the 24 IO ports surrounding (six per side) the chip can be wired to connect together multiple HyperX chips.

b. Integrated Software Development Environment

Programming the HyperX entails writing an ANSI C code, which defines the parallelism in the algorithm through the use of the industry standard Message Passing Interface (MPI) protocol. Note that, contrary to the IBM Cell or the Nvidia Tesla, no FORTRAN 2003 compiler for high performance numerical computations is yet available. The Coherent Logix integrated software tools automatically assign individual tasks to PEs, and create routing to support the movement of data between PEs. Tasks may be both parallelized and pipelined across multiple cores, while DMAs are controlled explicitly in software. The latter capabilities provide opportunities for designing the flow of program execution to meet resource constraints and programming requirements.

c. Power Management

The current version of the HyperX architecture provides the option to power down quadrants of the PE grid that are unneeded by a designed application. As a result, programs can be optimized with respect to energy usage (of the chip) as well as the computational speed and memory bandwidth usage. Wake-up signals can be triggered by external events, such that the processor may be shutdown during period of computational idle time.

II.2 The Cell Broadband Engine

The Cell multicore architecture is the product of five years of intensive R&D efforts undertaken in 2000 by IBM, Sony, and Toshiba [6]. Results reported in this paper refer to the PXC 8i (third generation) release of the processor, which is implemented on the QS 22 blades that utilize an IBM BladeCenter TM H. The PXC 8i model includes one multi-threaded 64-bit PowerPC processor element (PPE) with two levels of globally coherent cache, and eight synergistic processor elements (SPE). Each SPE consists of a processor (SPU) designed for streaming workloads, local memory, and a globally coherent DMA engine. Emphasis is on SIMD processing. An integrated high-bandwidth element interconnect bus (EIB) connects the nine processors and their ports to external memory and to system I/O. Details on the design parameters of the PXC 8i are well documented [9-11] and will not be repeated here. Note that both FORTRAN 2003 and C/C++ compilers for multi-core acceleration under Linux (i.e., XLF and XLC) are provided by IBM.

II.3 The NVIDIA Tesla Processor

The NVIDIA Tesla C1060 GPU [7] used in our system operates in conjunction with an Intel Xeon E5530 2.4 GHz CPU that accesses 6 GB of ECC DDR3 1066 MHz RAM. The operating system is the 64-bit edition of Windows XP. Our algorithms are programmed in CUDA FORTRAN using a compiler provided by the Portland Group Inc [12].
NVIDIA GPUs exploit the CUDA architecture [13], which, for the Tesla, is built in terms of an array of 30 Streaming Multiprocessors (SMs). Each SM consists of eight scalar processor cores. This results in a total of 240 cores for the Tesla. In addition, an SM includes two special units for transcendental functions, a multithreaded instruction unit, and 16 KB of shared memory. There are 4 GB of global RAM available to the GPU, and its clock runs at 1.296 GHz.

From a computational perspective, the fundamental underlying paradigm is the concept of scalar thread. Threads are grouped in blocks that execute concurrently on one SM with zero overhead [13]. Massive parallelism is achieved in terms of a kernel grid comprising the thread blocks. As any thread block terminates, a new block is launched on the first available (vacated) multiprocessor.

III. Inversion of Spatio-Temporal Covariance

The formal spatio-temporal detection problem can be stated as follows. Let \( r(i) \) denote the signal received, \( s(i) \) the unknown signal source, and \( \eta(i) \) the additive noise at the \( i \)-th element of a sensor array comprising \( N_s \) elements. Given a combined (stacked) array observable \( r \), i.e., a vector \( r = [(r(1)^T, r(2)^T, ..., r(N_s)^T)^T \) of \( N_t = N_s N_s \) elements. Given \( N_t = N_s N_s \) complex data, we must choose between two hypotheses \( \mathcal{H}_0 \) and \( \mathcal{H}_1 \), such that

\[
\mathcal{H}_0 \text{ (source absent): } r = \eta \tag{1}
\]
\[
\mathcal{H}_1 \text{ (source present): } r = s + \eta \tag{2}
\]

The likelihood ratio test is a central mechanism for making a decision between two hypotheses. In practice, this statistical test uses the natural logarithm of the ratio of the probability densities of the observable \( r \) under hypotheses \( \mathcal{H}_1 \) and \( \mathcal{H}_0 \), respectively [4]. Thus, it is usually referred to as the log likelihood ratio (LLR) \( L \). To compute \( L \), one must first derive expressions for these probability distributions, which involves the integration of complex Gaussian random processes. Omitting the propagation transfer matrices [5],

\[
L = \mathbf{z}^* \mathbf{A}^{-1} \mathbf{z} - \log_\odot (|\Sigma A|) \tag{3}
\]

where \( \mathbf{A} = \Sigma^{-1} + \mathbf{H}^{-1} \), \( \Sigma \) represents the covariance matrix of the signal, and the full spatio-temporal covariance matrix of the noise is denoted as \( \mathbf{H} \). Here, spatial components refer to sensors, while temporal components refer to measured samples. Specifically, matrix \( \mathbf{H} \) has the following structure:

\[
\mathbf{H} = \langle \eta \eta^* \rangle = \begin{bmatrix}
\mathbf{H}_{11} & \mathbf{H}_{12} & \cdots & \mathbf{H}_{1N_t}
\mathbf{H}_{21} & \mathbf{H}_{22} & \cdots & \mathbf{H}_{2N_t}
\vdots & \vdots & \ddots & \vdots
\mathbf{H}_{N_t1} & \mathbf{H}_{N_t2} & \cdots & \mathbf{H}_{NN_t}
\end{bmatrix}
\tag{4}
\]

\( \mathbf{H}_{ii} \) denotes the noise covariance matrix associated with the signal detected at sensor \( i \), and \( \mathbf{z} \) is a vector of “twice whitened” data, defined as

\[
\mathbf{z} = \mathbf{H}^{-1} \mathbf{r} \tag{5}
\]

This expression of \( \mathbf{z} \) (introduced in [5] to avoid replica whitening during target search), is distinct from the conventional expression for whitening data that is written \( \mathbf{z} = \mathbf{H}^{-1/2} \mathbf{r} \). The inversion of \( \mathbf{H} \) has complexity of order \( \mathcal{O}(N_s^3 N_t^3) \). We now show that by exploiting features in the spatially additive noise in conjunction with the FFT, we can reduce this complexity to \( \mathcal{O}(N_s N_t^3) \); given that \( N_s \ll N_t \), this reduction in the cost of spatio-temporal data whitening can be very significant.

III.1 Efficient Algorithms for Twice-Whitening

In order to design efficient algorithms to compute \( \mathbf{H}^{-1} \), we exploit the specific spatio-temporal organization of array ambient noise data. In particular, we observe that spatially diffuse noise is wide-sense stationary. This imposes the restriction that the submatrices of \( \mathbf{H} \) be Toeplitz, and that the full matrix be block Toeplitz. Note that this approximation excludes contributions from discrete interferers to the sources of noise in Eqs. (1-2). Then, the spectral theorem yields the eigenvalues’ matrix \( \mathbf{A}^{ij} \) for each \( N_s \times N_s \) submatrix \( \mathbf{H}^{ij} \), for \( i, j = 1, \ldots, N_s \). The Fourier factorization of the full covariance matrix \( \mathbf{H} \) can then be written

\[
\mathbf{H} = \begin{bmatrix}
\mathbf{F} & 0 & \cdots & 0 \\
0 & \mathbf{F} & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & \mathbf{F}
\end{bmatrix} = \hat{\mathbf{F}} \hat{\mathbf{A}} \hat{\mathbf{F}}^*
\]

\( \hat{\mathbf{F}} \) is just the \( N_s \)-point inverse Fourier transform (in unitary representation) applied to the time series at each of the \( N_s \) array elements. The matrix \( \hat{\mathbf{A}} \), of size \( N_s \times N_s \), is band-diagonal, in the sense illustrated on Figure 2.

![Block structure of the covariance matrix](image)
\[ H^{-1} = (\hat{F} \Omega \mathbf{K} \hat{\Omega}^T \hat{F}^*)^{-1} = (\hat{F}^*)^{-1} (\hat{\Omega}^*)^{-1} (\mathbf{K})^{-1} (\hat{\Omega})^{-1} \hat{F}^* \]  

(7)

\[ \mathbf{y} = \mathbf{\Omega}^T (\hat{F}^* \mathbf{z}) \]  

(9)

\[ \mathbf{d} = \mathbf{\Omega}^T (\hat{F}^* \mathbf{r}) \]  

(10)

However, since \( \mathbf{K} \) is block-diagonal, \( \mathbf{K}^{-1} \) can now be computed on a frequency by frequency basis, by inverting each block \( \mathbf{K}^{kk} \). As can readily be seen, this methodology will reduce the complexity of inverting the full space-time covariance matrix of size \( \mathbf{N}_s \times \mathbf{N}_s \) to one of inverting \( \mathbf{N}_s \) spatial covariance matrices. Each of these matrices is of considerably smaller size \( \mathbf{N}_s \times \mathbf{N}_s \). For \( k = 1, \ldots, \mathbf{N}_s \), we need to solve

\[ \mathbf{K}^{kk} \mathbf{y}^k = \mathbf{d}^k \]  

(8)

where \( \mathbf{y}^k \) and \( \mathbf{d}^k \) represent \( \mathbf{N}_s \times 1 \) ordered partitions of the transformed vectors

\[ \mathbf{y} = \mathbf{\Omega}^T (\hat{F}^* \mathbf{z}) \]  

(9)

\[ \mathbf{d} = \mathbf{\Omega}^T (\hat{F}^* \mathbf{r}) \]  

(10)

respectively. Eqs (9-10) are carried out first; thereafter, Eq (8) is solved via Cholesky decomposition. This result can then be permuted and inverse Fourier transformed in order to obtain the twice-whitened signal vector \( \mathbf{z} \).

III.2 Program Design

Our implementation of twice whitening on the hx3100 organizes the algorithm into multiple, pipelined and parallelized programs occupying 17 of the 100 available PEs. Refer to Figure 4 for locations of the programs described below. First, prior to any data processing, input data is loaded into off-chip DRAM memory banks. Here, the data is single-precision, complex-valued signal vectors stored in sample-consecutive order in DRAM 1. At runtime, program IOPE 1 fetches one input vector from DRAM 1 and copies the data onto the chip via a series of 4 routing programs labelled RPEs. The latter manage and synchronize the input data for our complex-to-complex FFT program.

Upon completion of the FFT of one input vector, the RPEs trigger IOPE 2 to store the transformed data off-chip in DRAM 2. Storage of the intermediate results is required to free on-chip memory for processing the FFT of the next input vector. The copying process occurs directly from the DMRs used by the FFT cores and, specific to our algorithm, requires a substantial amount of processing time. This overhead results from our current use of a strided write operation in which each complex-valued sample is written to DRAM with a stride of \( \mathbf{N}_s \). The purpose of this operation is to perform the permutation denoted by \( \mathbf{\Omega} \) in Figure 6.

Once all \( \mathbf{N}_s \) input vectors have been transformed and written to DRAM 2, IOPE 2 changes state and begins to fetch the permuted, length- \( \mathbf{N}_s \) vectors that represent the partitions \( \mathbf{d}^k \) defined in Eq. (10). IOPE 3 fetches a pre-computed Cholesky decomposition from DRAM 3. The latter is represented by the \( \mathbf{N}_s ((\mathbf{N}_s-1)/2 \) elements of an upper triangular, complex-valued matrix obtained by offline decomposition of the matrix \( \mathbf{K}^{kk} \). Future versions of this program will implement the Cholesky decomposition of each such matrix alongside the transformation and permutation of the signal vectors. Both sets of data are moved to the DMRs of the whitening program (LPE). Within this program forward elimination and back substitution solve Eq. (8), both requiring \( \mathcal{O}(\mathbf{N}_s^2) \) operations to complete. The output of this program is written back to DRAM 4 via IOPE 4. Again, a strided-write operation is used, with the \( \mathbf{N}_s \) components (samples) spaced by \( \mathbf{N}_s \) positions; once all \( \mathbf{N}_s \) solutions are obtained, DRAM 4 stores, in consecutive order, the \( \mathbf{N}_s \) partitions of \( \mathbf{\Omega} \mathbf{y} \) obtained from Eq (9), i.e., the FFT of the twice-whitened vector \( \mathbf{z} \).

An important consideration is the synchronization between individual PEs to ensure continuity and correctness of data movement. The 8 KB memory available from each DMR necessitates precise accounting for the size, location, and timing of each data segment. This inter-core communication is performed using a special purpose API written for the C language. For example, blocking variants of DMA send and receive functions and provide a basic communication mechanism for data synchronization, and additional mechanisms are available for reading and writing to DRAM.

Fig 3. Transformation of the band-diagonal matrix \( \Lambda \) by the orthogonal permutation matrix \( \Omega \) yields the block-diagonal matrix \( \mathbf{K} \), where each diagonal block \( \mathbf{K}^{kk} \) is of size \( \mathbf{N}_s \times \mathbf{N}_s \).

Fig 4. Layout of the STTW program on the hx3100 processor. Boxes circumscribe PEs and DMRs for each individual program.
The placement and routing of individual programs, e.g., the IO servers or the FFT cores, with respect to the layout of available cores is also an important consideration for performance. While this level of detail can be programmed explicitly, the HyperX software development environment provides a global optimizer and scheduler to perform placement and routing. For example, routing constraints ensure the IO server programs are placed adjacent to the off-chip IO ports and that on-chip DMA routes do not collide, e.g., when transferring data between PEs.

IV. Computational Results

As implemented, the full program processed a series of 8192-complex-point input vectors, each representing a noisy sinusoidal typical of a monotone acoustic source collected against a diffusive (thermal) background. At a sampling rate of 64 KHz, this amounts to ~125 ms of data. When running on the hx3100 processor, input vectors are loaded into the off-chip DDR2 memory banks. Vectors are stored in de-interleaved, bit-reversed order, with a single input vector representing 64 KB of memory. Eight DMA transfers are required to load a collection of 8 on-chip DMRs. This IO service requires 1 core for managing requests to the DDR2 and, in a double buffering approach, 3 neighboring DMRs to optimize timing of the data transfer.

An 8192-point, complex-to-complex decimation-in-time FFT is used to transform the input data. Each instance employs 8 PEs, as shown in Figure 4. The FFT algorithm requires bit-reversed-order input data and reordering of the input is done prior to loading the DRAM. The input and output servers are synchronized to avoid overwriting data stored in the FFT processing cores.

We have implemented the above on the HyperX hx3100, and we have profiled the program elements using the Integrated Software Development Environment. The results below refer to version 3.0.1 of the HyperX ISDE tools for the case of \( N_e = 8 \) and \( N_e = 8192 \) complex samples. The number of cycles, normalized to the relevant input type, is shown for each program in Table 1.

Table 1. Program resources used, including PEs, DMRs, and cycles per iteration of each program element.

<table>
<thead>
<tr>
<th>Program</th>
<th>PEs</th>
<th>DMRs</th>
<th>Cycles/iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOPE 1</td>
<td>1</td>
<td>4</td>
<td>37,494</td>
</tr>
<tr>
<td>RPE's</td>
<td>4</td>
<td>16</td>
<td>N/A</td>
</tr>
<tr>
<td>FPE's</td>
<td>8</td>
<td>18</td>
<td>137,741</td>
</tr>
<tr>
<td>IOPE 2 (write)</td>
<td>1</td>
<td>4</td>
<td>565,749</td>
</tr>
<tr>
<td>IOPE 2 (read)</td>
<td>1</td>
<td>1</td>
<td>136</td>
</tr>
<tr>
<td>IOPE 3</td>
<td>1</td>
<td>2</td>
<td>358</td>
</tr>
<tr>
<td>LPE</td>
<td>1</td>
<td>4</td>
<td>4,384</td>
</tr>
<tr>
<td>IOPE 4</td>
<td>1</td>
<td>3</td>
<td>539</td>
</tr>
<tr>
<td>TOTAL</td>
<td>17</td>
<td>52</td>
<td>42,014,925</td>
</tr>
</tbody>
</table>

For example, IOPE 1 requires 37,494 clock cycles to read one input vector and synchronize transfer of that vector to the RPEs memory space. This accounting of clock cycles highlights the processing bandwidth associated with each program; data transfer is not computationally complex, but a moderate amount of time is needed to bring the data onto the chip. Regarding IOPE 1, the reported cycles include sending data to the RPE's, while the cycles reported for the FPEs incorporate the read operation from the RPE memory space. IOPE 3 reads the transformed and permuted vectors \( d^k \) from DRAM 3; recall the latter are identified by the sample (or, equivalently, frequency) index \( k \) and there are a total of 8192 vectors to process. The other program elements have similar interpretations of the reported cycle counts.

Table 2 reports the total power consumption with respect to clock frequency and voltage.

<table>
<thead>
<tr>
<th>Voltage(V)</th>
<th>Tunable PE Clock Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
</tr>
<tr>
<td>1.00 V</td>
<td>914.29</td>
</tr>
<tr>
<td>0.95 V</td>
<td>865.95</td>
</tr>
<tr>
<td>0.90 V</td>
<td>820.09</td>
</tr>
</tbody>
</table>

Fig 5. In IOPE 2, input vectors are sequentially transformed and permuted using an \( N_e \)-strided write to DRAM. Consecutive memory is from left to right. As a result, the vector \( d^k \) read directly from DRAM 2, can be immediately used for the twice-whitening operation via \( \mathbf{K}^{kk} \mathbf{y}^k = d^k \).

IOPE 2 represents the most costly operations in this implementation. As stated above and illustrated by Figure 8, IOPE 2 is performing a sample-by-sample strided write of the FFT output. Many samples (8192) must be processed this way and the cost associated with strided writes to DRAM is high. More important, the IOPE 2 write operation represents a significant bottleneck in this implementation as the write operation takes significantly more time than the FFT processing. However, the size of the processed data set (8 × 8192 samples = 512 KB) prohibits performing transposition within on-chip memory space. The cost of the strided-write operation is constant with respect to the number of input vectors. This problem will be addressed in future work.

Our present implementation omits concurrent calculation of the Cholesky decompositions for permuted and transformed noise covariance matrices \( \mathbf{K}^{kk} \). Note that those calculations utilize a method identical to the computation of \( \mathbf{y} \) (in timing and complexity), and that the decomposition of each \( \mathbf{K}^{kk} \) could be easily accommodated within the LPE provided \( N_e \) is not too large (~35).

Table 2. Power consumption with respect to clock frequency and voltage.
time-to-solution of the program. Based on the total cycles reported in Table 1, a range of 84.2 ms at 500 MHz to 210 ms at 200 MHz is observed. Lowering the voltage, from 1.0 V to 0.95 V to 0.9 V, provides a corresponding decrease in power. A two-fold increase in power occurs between the case of 0.90 V at 200 MHz and the 1.0 V at 500 MHz. Thus, depending on the performance requirements, STTW on the hx3100 can satisfy a range of power and timing constraints.

V. Alternative FFT Implementations

As evidenced by Eqs (7, 9, 10) an efficient FFT lies at the heart of the ambient noise covariance matrix inversion paradigm. In that context, results of two alternative schemes for its implementation on the IBM Cell and the Nvidia Tesla are included in this Section. The mathematical foundations of the FFT are discussed in detail in Van Loan’s seminal monograph [14]. Exploiting the capabilities of SIMD processors to improve the performance of the FFT has long been of interest to the applied mathematics, signal processing, and computer science communities. Most of the latest reported innovations attempt to achieve optimal device-dependent performance by optimizing cache utilization or vectorizing operations carried out on a single data sequence. We define such paradigms as inline vectorization. Recently [15], motivated by the fact that acoustic time-sampled array data can naturally be partitioned across multiple SIMD-capable cores, we have addressed a complementary question. We proposed an algorithm where M 1D data arrays, each of length N, would be Fourier-transformed concurrently by a single IBM Cell SPE core. This resulted in 8M arrays that could be handled simultaneously by the Cell processor, with each core exploiting its own SIMD capability. We defined this paradigm as transverse vectorization.

V.1 Transverse Vectorization on the IBM Cell

Our algorithms were implemented using programs written in a mixed language framework (FORTRAN 95/2003 and C) based upon the IBM XLF and XLC compilers for multicore acceleration under Linux. They are specifically exploiting the intrinsic data structures and functions available for SIMD operation. Our results are illustrated in Figure 6.

The results are for vector lengths of 1024 complex samples, SIMD transverse vectorization of 4, and batches of 64 vectors per core. Our method was shown to outperform, by at least a factor of two, the fastest results from competing leading edge methods published to date in the open literature [9].

V.2 Transverse Vectorization on the Nvidia Tesla

The situation is somewhat different with the NVIDIA Tesla, where the SIMD concept does not directly apply. Rather, the CUDA architecture exploits a single-instruction-multiple-thread (SIMT) concept [13]. Each thread block executing on an SM is partitioned into groups of 32 threads (called warps) that are scheduled by the SIMT unit to execute concurrently. Under SIMT, each thread from a warp is assigned to one of the scalar processor cores belonging to the SM. Threads composing the warp start at the same program address, but are nominally free to branch and execute independently. At every instruction select time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads in the warp. If threads in a warp diverge via data-dependent conditional branching, the warp serially executes each branch path taken, while disabling threads that are not on that path. Finally, when all paths complete, the threads converge back to the same execution path [13]. Since a warp executes one common instruction at a time, the highest efficiency is achieved when all 32 threads in a warp agree on their execution path. This is precisely what happens under the transverse vectorization construct, which we had defined in the context of the Cell processor.

Our implementation comprises two components. A code that runs on the host CPU and a kernel that runs on the GPU and is invoked by the host. In practice we run a set of identical kernels on a one-dimensional grid partitioned into thread blocks. These blocks get assigned to different SMs of the GPU as scheduled by the hardware. Because of the latencies associated with data retrieval from the GPU’s global memory, there is a strong incentive to maximally exploit all banks of shared memory, as well as the constant memory.

In terms of specific algorithms, we report below our initial results for a radix-8 Stockham scheme [14].

<table>
<thead>
<tr>
<th>NX</th>
<th>BATCH</th>
<th>TF</th>
<th>TB</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1048576</td>
<td>0.00373</td>
<td>0.00373</td>
<td>33.69</td>
</tr>
<tr>
<td>64</td>
<td>131072</td>
<td>0.00946</td>
<td>0.00959</td>
<td>26.38</td>
</tr>
<tr>
<td>512</td>
<td>16384</td>
<td>0.00875</td>
<td>0.00991</td>
<td>40.46</td>
</tr>
<tr>
<td>4096</td>
<td>2048</td>
<td>0.01225</td>
<td>0.01219</td>
<td>41.18</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
<td>0.02027</td>
<td>0.02007</td>
<td>49.89</td>
</tr>
<tr>
<td>4096</td>
<td>8192</td>
<td>0.04564</td>
<td>0.05460</td>
<td>39.81</td>
</tr>
</tbody>
</table>

Table 3. FFT timings on Tesla C1060 (preliminary). NX = length of vector; BATCH = number of vectors; TF = forward transform time; TB = inverse transform time.
In the actual subroutine, we have implemented loop unrolling. Moreover, even though complex numbers are supported both by the NVIDIA’s CUDA C and PGI’s CUDA FORTRAN compilers, we have split the real and imaginary components to achieve better alignment in memory fetches. This splitting also enables the use of a single set of weights. It is important to note that the results reported in Table 3 were obtained without invoking the compiler optimization options (due to yet unresolved data transfer error messages in PGI’s CUDA FORTRAN 10.8 compiler). Hence, there is a substantial room for performance improvement. Also, no attempt was made at this stage to optimize the data partition between global and shared memory. As such, this implementation does not yet reach the performance reported by Govindaraju et al [16]. On the other hand, a direct use of CUDA’s FFT library [17] produces the following results.

![GILOPS vs log2(N00)](image)

**Fig 7.** Throughput performance of CT radix-2 scheme on our system

### V. Summary and Conclusions

To achieve the real-time and low power performance required for maritime sensing and other computationally demanding applications, many existing algorithms may need to be revised and adapted to the emerging revolutionary computing technologies. Novel hardware platforms of interest to naval applications include the IBM Cell, the Coherent Logix HyperX, and the NVIDIA Tesla (Fermi) devices.

In this article, we have developed, implemented, and demonstrated a novel algorithm for spatio-temporal twice whitening on the ultra-low power hx3100 processor. Our implementation exploits the structure of the spatially diffuse noise covariance matrix to reduce the computational complexity by a factor of $N_s^2$, amounting to several orders of magnitude. Moreover, we demonstrated that this critical signal processing task can be performed in real time on the hx3100. This processor provides a platform capable of handling adaptive computations through real-time reconfigurability. The high throughput and low-power features of our realization suggest new signal processing opportunities for acoustic energy detectors in maritime sensing applications. Because of the essential role the FFT plays in such applications, we also included results for the transverse vectorization paradigm on the IBM Cell. This algorithm was recently shown to outperform, by at least a factor of two, the fastest results from competing leading edge methods published to date in the open literature. We also included preliminary results for a radix 8 Stockham FFT implementation on the Nvidia Tesla. The recent release by PGI of the high-performance CUDA FORTRAN compiler for the NVIDIA Tesla opens, via mixed language (C and FORTRAN) programming an optimal framework for ultra fast future implementations. Such a framework would fully exploit the intrinsic array language, compiler optimization and numerical capabilities of FORTRAN in conjunction with the DMA and system capabilities of C.

Finally, we believe that in the longer term, the emergence of multicore devices will enable the implementation of novel, more powerful information–processing paradigms that could not be considered heretofore.

### References

7. www.nvidia.com
8. M. Stolka (Coherent Logix), personal communication; see also: www.coherentlogix.com