Defence Science Journal, Vol. 74, No. 5, September 2024, pp. 755-762, DOI : 10.14429/dsj.74.19505
© 2024, DESIDOC

A High-Performance Parallel Approach to Delay Sum Beamformer in a Homogeneous Multicore Environment

M. Prakash Narayanan*, G. Vijay Gopal and R. Rajesh

DRDO-Naval Physical and Oceanographic Laboratory, Kochi - 682 021, India

**E-mail: prakashnm.npol@gov.in*

ABSTRACT

A Cache-Aware Beamformer (CABF) algorithm for the DAS beamformer in a homogeneous multicore processor environment is presented. The context of the proposed algorithm is established by discussing the case for a refined multicore implementation of the beamformer algorithm for a sonar application. The algorithm is designed, implemented, and compared to a regular pthread multicore implementation and a standard OpenMP-based implementation, using arithmetic intensity as the metric. FMA implementations of the algorithms are carried out, and the CABF algorithm is shown to achieve a better arithmetic intensity. A 6000-element array is designed with a simultaneous forming of 200 beams to test the efficacy of cabf in a multicore platform. The results show a 73 % increase in GFLOPS for FMA operations. The performance of the beamformer algorithm for different data sizes is studied, and on average, a 36 % improvement in computational performance is achieved compared to the OpenMP-based implementation.

Keywords: Underwater acoustics; Delay sum beamformer; Multicore processing; OpenMP; Sonar signal processing

NOMENCLATURE

CABF	: Cache aware beamformer
DAS	: Delay and sum
GFLOPS	: Giga floating point operations
FMA	: Fused multiply add
OS	: Operating system
FPGA	: Field programmable gate array
DSP	: Digital signal processor
CPU	: Central processing unit
GPU	: Graphics processing unit
OpenMP	: Open multi-processing
MPI	: Message passing interface
POSIX	: Portable operating system interface
DRAM	: Dynamic random access memory
CPN	: Computational process networks

1. INTRODUCTION

Advances in computing technology have enabled the design of complex beamforming arrays with a large number of elements, which is essential for detecting underwater targets in an increasingly deceptive environment. Underwater targets have become less noisy[1], making passive detection more difficult. Ambient noise conditions have increased due to navigation, oil exploration, and naval activities[2-3], which adversely affect sonar performance. As a result, sonar designers resort to bigger arrays as one of the design parameters. Design considerations are discussed in detail in literature[4-6]. Arrays with thousands of sensors are now common, making it extremely computationally intensive to process and extract target information.

Received : 16 October 2023, Revised : 16 January 2024
Accepted : 02 February 2024, Online published : 19 September 2024

With the easy availability of multicore processors, computationally intensive processing for big arrays has become a reality. To achieve optimal performance, users must closely examine the hardware architecture of processors. With Embedded Linux as the OS, a careful analysis of the memory architecture can significantly improve application performance[7-9].

In an array with a multitude of sensors, tuning for performance becomes important. The beamformer is the first stage in array signal processing and is the most computationally and memory-intensive operation[10]. Different types of beamformers are discussed in the literature[10-11] and are tuned for different applications[12]. High-throughput, compact, and low-power systems typically use FPGA[13-14], while moderately intense systems useDSP[15]. With the advent of versatile processors yielding multiple cores, processor-based (CPU) and heterogeneous (CPU/GPU) approaches are used.

Our work focuses on the time domain DAS beam former for Sonars in a homogeneous CPU environment. There is a potential for compact, low-power, and cost-effective solutions for very large arrays on multicore processors. This will enable the realization of applications like sonar, radar, and medical imaging on readily available multicore machines. Previous investigations analysed the effects of the hardware architecture of the CPU[16,5], proposing methods to improve the usage of computational resources.

To further improve performance in a multicore environment, we propose the CABF algorithm. Computationally intensive parts are identified, and the proposed algorithm focuses on the reuse of data fetched from main memory (DDR).

Finally, we compare the proposed beamformer algorithm with a regular multicore implementation, as well as a standard

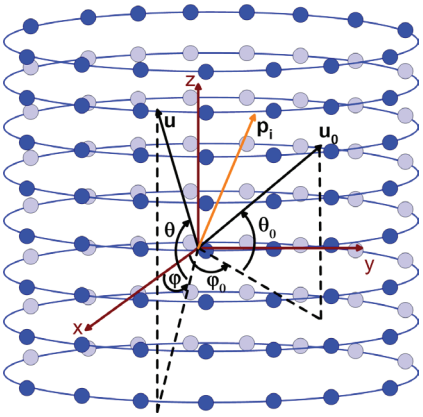


Figure 1. Cylindrical array with beam steering in an arbitrary direction with steering angle (φ_0,θ_0) in bearing and elevation.

Open MP-based implementation on an Intel Xeon platform. The results show an overall improvement of 26 % in GFLOPS for the proposed algorithm over the pthreaded version and 36 % in GFLOPS over the Open MP-based implementation.

1.1 Contributions

The contributions of the work include:

- Analysis of the beamformer algorithm to identify the places of performance improvement, considering the processing resources available in multicore processors.
- Efficient reusing of data fetched from the main memory.
- Parallel beamformer implementation on the 6000-element cylindrical sensor array.
- 36 per cent improvement over a standard OpenMP implementation, and 26 % over the pthreaded implementation.

Section 2 will present the theory behind the DAS beamformer, focusing on identifying the steps to make the proposed algorithm. Section 3 will present the implementation steps on a multicore processor leading to the proposed CABF algorithm. Section 4 will discuss the results of implementation, and Section 5 concludes the findings of the work.

2. DELAY SUM BEAMFORMER

2.1 Introduction to Beamformer

The time DAS beamformer calculates and compensates for the delay incurred by a plane wave front that impinges on the elements and adds the individual sensor signals, producing a beam in a specified direction. In Fig.1 the beam is formed in the direction of vector u_0. The position vector of the i^{th} sensor is p_i. (φ,θ) represent the bearing and elevation angles. Assuming a plane wave propagating in the direction of u, the sinusoidal signal of frequency f is given by:

$$s(n) = e^{j2\pi fn/f_s} \tag{1}$$

The time-delayed signal in the i^{th} sensor is given by,

$$x(n) = e^{j\left(\frac{2\pi fn}{f_s} + k\mathbf{p_i}.\mathbf{u}\right)}, i = 1,2,...N_b \tag{2}$$

where, N_b is the number of sensors, k=2πf/c –wave number, f_s-Sampling frequency, and

c-Velocity of signal propagation.

Summing the signals across N_b sensors yields the beamformed output.

$$y(n) = \sum_{i=1}^{N_b} x_i(n) = e^{j2\pi fn/f_s} \sum_{i=1}^{N_b} e^{jk\mathbf{p_i}.\mathbf{u}} \tag{3}$$

To get the beam steered in an arbitrary direction u_0, (3) becomes,

$$y(n) = e^{j2\pi fn/f_s} \sum_{i=1}^{N_b} e^{jk\mathbf{p_i}.(\mathbf{u}-\mathbf{u_0})} \tag{4}$$

In (3) and (4), the term $e^{j2\pi fn/fs}$ represents the plane wave signal (1) from the target, and the second term represents the beam pattern. $p_i.(u-u_0)$ gives the delay required for the i^{th} sensor for the beam to form in the direction of u_0. The time delay computation must be precise so that the signals are compensated exactly. Equation (5) gives the expression to compute the time delay for a signal arriving from (φ,θ).

$$\tau_i = -\frac{1}{c}[\cos\theta\cos\varphi.p_{x_i} + \cos\theta\sin\varphi.p_{y_i} + \sin\theta.p_{z_i}] \tag{5}$$

2.2 Beam Steering, Time Delay, and Weighting

The beam is steered in a specific direction by delaying the sensor signals and adding multiple sensors. For precise positioning of beams, the time delays are not integer multiples of samples. Hence, after obtaining the integer sample delays (by selecting samples), it is necessary to interpolate the residual fractional time delays. This makes fractional interpolation a crucial first step in realizing the DAS beamformer. Together with windowing w_i, for side lobe reduction, (3) will be modified as follows:

$$y(n) = \sum_{i=1}^{N_b} (w_i.x_i(n+\Delta_i)) \tag{6}$$

where, Δ_i is the fractional delay for the i^{th} channel. More details on interpolation can be found in literature[17-20].

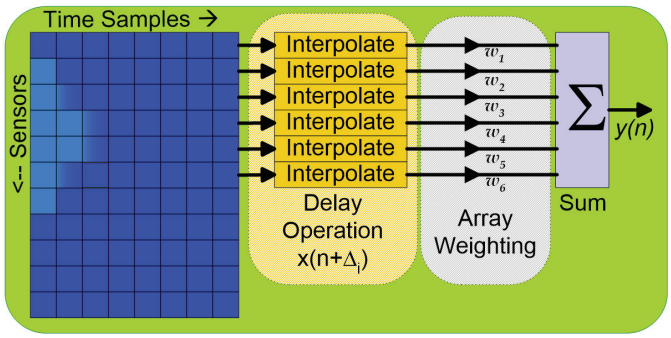


Figure 2. Block diagram of DAS Beamformer(6).

2.3 Analysis of Computational Requirements

Figure 2 depicts the implementation of Eqn. (6). The time samples are interpolated to achieve the precise fractional delay to form a beam in the direction (φ,θ). Array weighting is applied to the individual sensor signals before summing with integer delays, to obtain the beam time series.

Table 1 summarizes the computational requirements of the DAS beamformer/sample. Since N_b is typically of the order of 2000 (example design in Section 3.5), and 200 beams are formed simultaneously at a sampling rate of 32kHz, the

Table 1. Computational requirements for a single beam per time sample.

	Multiplications	Additions
Interpolation	6 * Nb	5 * Nb
Array weighting	Nb	-
Sum output	-	Nb - 1
Total (DAS)	7 * Nb	6 * Nb - 1

computational requirements are high and require an optimised implementation.

3. REALISATION OF MULTICORE BEAMFORMER

3.1 Literature Survey

The most common approach to the realisation of beamformer is to use OpenMP/MPI libraries or a hybrid approach that combines POSIX threads[21] with OpenMP[22-23]. OpenMP-based multicore implementation is discussed in ref[24] for an image processing system. Implementing the sonar processing on workstations connected over a network using POSIX threads is discussed in a paper by Allen.GE and Evans. BL[25]. In reference[26] a comparison of frequency domain beamformers with Open MP and CPN, which is a custom model for signal processing applications, is carried out. In one of the papers[27] details of a multi-beam sonar implementation using FPGA and TI-DSP is given. The realization of a distributed beamformer for sonar signal processing is given in reference[28], which utilizes workstations distributed over a network. However, studies on the optimization of beamformers focused on a multicore processing environment are not available in the literature. In our work, a General Purpose Processor-based POSIX thread implementation is carried out, with suggested improvements, and compared against Open MP standard-based implementation.

3.2 Performance Measure: Arithmetic Intensity

Arithmetic Intensity (AI) is used as the metric to measure the efficiency of data utilization. AI is the ratio of work (W), which is the FLOPS (Floating Point Operations Per Second), carried out by the CPU to memory traffic (Q), in *bytes/s.*

$$AI = \frac{W}{Q} \tag{7}$$

This is a measure of computations performed per byte of data fetched from memory. This quantity is measured from all levels of memory (L1, L2, L3 Cache, and DDR) to observe the improvement in data reuse at each level.

3.3 Details of OpenMP Implementation

The pseudo-code for the single-core implementation of the beamformer is given in Algorithm 1. The inner loop runs for the total number of time samples(N). The first step in the computations is interpolation for fractional delays $x_i(\Delta_i)$, followed by array weighting $w_i.x_i(\Delta_i)$ and then adding $y(n)=\Sigma(w_i.x_i(n+\Delta_i))$ corresponding samples across sensors (Eqn. (6)).

Algorithm 1: Single core implementation

```
for j=0,B−1 do
    for i=0,Nb−1 do
        Coeff=Copy_Delay_Coefficients(j,i)
        for k=0,N−1 do
            Interpolating_Filter(k,Coeff)←x(i,k)(Δi)
            Weight(is)←wis.x(i,k)(Δi)
            n=integer_delay_for_(i, j)
            beamOut=Add(j,k)←∑wis.x(i,k)(n + Δi)
        end for
    end for
end for
```

The sensor loop runs for N_b sensors to obtain a beam in one direction. The beam loop runs for a B number of beams to be formed. OpenMP constructs are used to instruct the compiler to parallelize for loops. This is used to evaluate the performance of CABF.

3.4 Details of Multicore Implementation

The beamformer is implemented to use multiple cores by dividing the beams among them, using a multi-threaded approach with pthread (POSIX Thread). A straightforward

Algorithm 2: Multi-core Implementation

```
BeamformerThread()
threadID=getThreadID() {get thread number}
BT=totalBeams/numThreads
startBeam=BT*(threadID-1)
NS=numSensorsInArray
for j=startBeam;startBeam+BT−1 do
    startSensor=j*Ns
    i=startSensor%NS
    for l=0,Nb−1 do
        Coeff=Copy_Delay_Coefficients(j, i)
        for k=0,N−1 do
            InterpolatingFilter(k,Coeff)←x(i,k)(Δi)
            Weight(is)←wis.x(i,k)(Δi)
            n=integer_delay_for_(i, j)
            beamOut=Add(j,k)← ∑wis. x(i,k) (n + Δi)
        end for
        i=i+1
        i=i%NS
    end for
end for
```

extension is not an optimized solution [6], as it requires taking the cache architecture of the processor into account.

In the pseudo-code of Algorithm 2, the inner loop (lines 11-16) performs interpolation for the fractional time delays $x_i(\Delta_i)$, weighting, and adding of the data samples considering the integer delay requirements of each sensor, with an encapsulating sensor loop (9-19). The beam loop (6-20) differs in each core. The thread ID (line 2) identifies the beams to be processed by the threads (B_T-beams/thread). N_S is the total number of sensors in the array and the sensors participating in the beam are dynamically computed in the loop (variable i).

3.5 Cache-Aware Beamformer for Multicore Processors

In this section, we analyze the algorithm 2 implementation and identify areas to improve its performance. The CABF focuses on reducing memory transactions and reusing data from cache.

Figure 3 illustrates the arrangement of data in the memory of the processor. Sensor time samples are arranged in memory, with consecutive memory locations holding a sensor's time series. The consecutive sensors in bearing and elevation (green/yellow) are used for a set of beams. Each thread requires some overlap in the data (prologue and epilogue) which will be accessed by consecutive threads (blue in Fig. 3). For consecutive beams the sensor indices will be incremented in bearing by one. When the beam loop increments, a new set of sensor data is fetched from memory. However, only one subset of sensors in the elevation direction is different in the adjacent beam. This provides an opportunity for data reuse. The example design presented in [5] is used to illustrate this point.

Consider a cylindrical array with 30 elements in the vertical and200 in the bearing direction (6000 elements array). Let us assume that only 1/3rd of the sensors in the bearing participate in the beamformer. We take 64x30=1920($N_\varphi X N_\theta$) elements for the beamformer, where N_φ and N_θ are the sensors in the bearing and elevation for the beam. The time samples $x_i(n)$ (N=2048) arranged as shown in Fig. 3 are given to the beamformer thread. In Algorithm 2, as the beam loop increments, a set of 1920 sensor time series (1920x2048) will be fetched for computing the new beam. However, only 30 new sensors are there, not used in the previous beam. As the beam progresses, we can see a sensor is used in 64 beams and re-fetched each time. This can be avoided by repositioning the sensor and beam loop and carefully managing the loop variables and memory. Once sensor data is fetched it needs to be reused for all the N_φ=64 beams before it is discarded. Note that, the beam output data will be fetched multiple times to accumulate the sensor data. As multiple threads are spawned (here4), each thread will get 50 out of 200 beams. The smaller the number of beams, the more it is cacheable (depending on L3 capacity). This will enable faster access to the beamformer intermediate data and its reuse.

As the beam loop and sensor loops are interchanged, we need to split the computations into 3 separate sections to get the maximum advantage of reuse; prologue, steady state, and epilogue. In the epilogue, the first set of sensors (N_θ=30) will be used only in one beam, whereas the next 30 will be used in

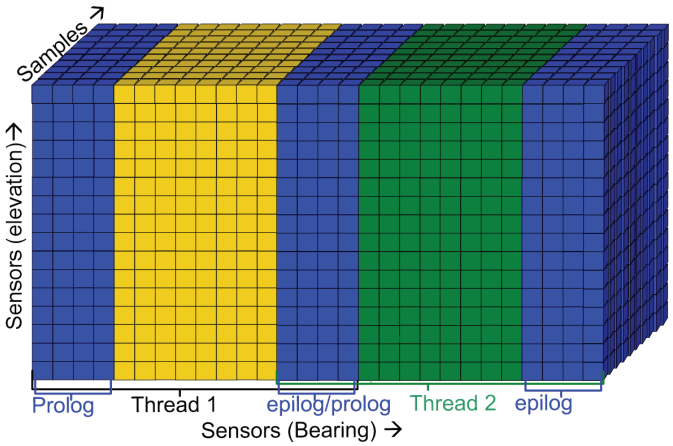


Figure 3. Data cube showing the memory arrangement of sensors time samples $x_i(n)$ in bearing and elevation. The blue portion shows the data that is shared between threads processing adjacent blocks of bearing.

Table 2. Memory Bandwidth for fractional interpolation in A2 and A3.

Data size (k Data)	Memory bandwidth (G bytes/s)							
	L1		L2		L3		DRAM	
	A2	A3	A2	A3	A2	A3	A2	A3
1	148	417	78	5.3	79	0.015	34	0.001
2	94	409	66	219	49	0.011	35	0.001
4	96	334	120	452	48	0.002	44	0.001
6	97	324	145	428	49	0.001	45	0.001
8	104	305	151	429	53	0.002	49	0.001
10	100	306	152	450	51	0.002	48	0.001

Table 3. Arithmetic intensity for fractional interpolation in A2 and A3

Data size	Arithmetic intensity (Interpolation)							
	L1		L2		L3		DRAM	
	A2	A3	A2	A3	A2	A3	A2	A3
1	0.88	0.88	1.65	62.8	1.65	2.5e4	3.78	8.0e5
2	0.88	0.88	1.24	1.63	1.69	3.4e4	2.32	9.1e5
4	0.88	0.88	0.69	0.65	1.72	1.5e5	1.91	9.4e5
6	0.88	0.88	0.58	0.66	1.73	1.9e6	1.87	1.2e6
8	0.88	0.88	0.61	0.62	1.74	1.6e5	1.83	1.3e6
10	0.88	0.88	0.59	0.59	1.74	1.1e5	1.83	1.1e6

Table 4. Memory bandwidth for sensor data addition with integer delay operation in A2 and A3.

Data size	Memory bandwidth (G bytes/s)							
	L1		L2		L3		DRAM	
	A2	A3	A2	A3	A2	A3	A2	A3
1	656	397	3.7	243	0.087	0.06	0.003	0.02
2	754	382	188	242	0.033	0.06	0.007	0.05
4	516	376	413	298	0.001	1.46	0.001	0.07
6	427	331	381	306	0.001	103	0.001	0.07
8	436	268	443	266	0.001	126	0.001	0.05
10	467	256	447	257	0.001	139	0.001	0.11

Algorithm 3: Cache aware beamformer implementation

```
1: BeamformerThread()
2: threadID = getThreadID()
3: Ns = numSensorsInArray {Sensors in array}
4: BT = totalBeams/numThreads
5: startBeam = BT * threadID
6: startSensor = Ns * startBeam
7: i = startSensor % Ns {Running Sensor Index}
8: beamTmp[0:BT-1][0:N-1] = 0.0
9: sensorIn[N] = 0.0 {Temp buffer for xi}
10: for l = 0,sensorPerCore − 1 do
11:     beamCnt=int(l / Ns)
12:     is=( l % Ns)+beamCnt*Ns
13:     sensorIn = copy xi ← xi(0 : N − 1)
14:     if beamCnt < prologue then
15:     for j = 0,beamCnt − 1 do
16:       Coeff = Copy Delay Coefficients (j, is)
17:       for k = 0,N − 1 do
18:         Interpolating Filter(k, Coeff) ← x(i,k)(Δi)
19:         Weight(is) ← wis.x(i,k)(Δi)
20:         n = integer delay for (i, j)
21:         beamTmp=Add(j,k) ← ∑wis. x(i,k)(n + Δi)
22:       end for
23:       is = is - Ns
24:     end for
25:     else if beamCnt < steadyState then
26:       for j = 0,BT − 1 do
27:         Coeff = Copy Delay Coefficients (j, is)
28:         for k = 0,N − 1 do
29:           Interpolating Filter(k, Coeff) ← x(i,k)(Δi)
30:           Weight(is) ← wis.x(i,k)(Δi)
31:           n = integer delay for (i, j)
32:           beamTmp=Add(j,k) ← ∑wis. x(i,k)(n + Δi)
33:         end for
34:         is = is - Ns
35:       end for
36:       else if epilogue then
37:         bmLpStart = beamCnt - Np + 1
38:         sensorInBeam = sensorInBeam-bmLpStart*Ns
39:         for j = bmLpStart,BT − 1 do
40:           Coeff = Copy Delay Coefficients (j, is)
41:           for k = 0,N − 1 do
42:             Interpolating Filter(k, Coeff) ← x(i,k)(Δi)
43:             Weight(is) ← wis.x(i,k)(Δi)
44:             n = integer delay for (i, j)
45:             beamTmp=Add(j,k) ← ∑wis. x(i,k)(n + Δi)
46:           end for
47:         is = is - Ns
48:       end for
49:     end if
50:       i = (i + 1) % Ns
51: end for
52: beamOutput (startBeam:startBeam+BT) = beamTm
```

beams 1 and 2. This progressively increases, and 64 beams are in a steady state. This portion of ramping up the data usage is called the prologue, and at the end, the reverse of this happens in the epilogue.

The pseudo-code of the above-explained modification is given in Algorithm 3. The sensor loop and beam loop are interchanged. Depending on thread ID, the beams to be processed are found in threads, and a temporary buffer for the beam output is allocated (lines 8 & 9), small enough to get cached as per processor architecture (*N* is selected accordingly). Corresponding sensors are found out (*start Sensor*, line 6). With beam count value the prologue (lines 14-24), steady-state (lines 25-35), and epilogue (lines 36-49) portions of the beamformer are identified. The beam loop does all computational functions required for the beamformer (interpolation, weighting, and adding sensors). Beam output is copied to DRAM in line 52.

3.6 Hardware Used for Evaluation

To evaluate the performance, an Intel Xeon W series multicore machine with a maximum clock speed of 3.7 GHz, and Linux kernel 5.15 OS is installed is used. It has 32 kB L1 & L2, and 20 MB L3 cache.

4. RESULTS AND ANALYSIS

The algorithms presented in Section III are implemented using vector-optimized IPP libraries from Intel. Fractional delay $x_i(\Delta_i)$ is implemented using the FIR filter function. Index generation for vector addition takes care of integer delay, giving the final beamformer output.

The results are collected with four threads spawned on as many cores. The performance is studied for different data sizes, and is presented with 'kilo floating point data', marked as 'k Data' in figures and tables. Table 2 shows the bandwidth in GBPS for each level of memory of the processor. For Algorithm 3, the bandwidth of L1 is higher, but it decreases as the data size increases (Table 2). The bandwidth of L2, which is initially small, increases gradually. However, the bandwidth of L3 and DRAM remains very low, as seen in Table 2. This implies, once the data is copied to a temporary buffer, the buffer is cached. In contrast, Algorithm 2 has higher data rates from DRAM for all cases, as the data is progressively fetched to L1 (Table 2). The corresponding AI for all levels of memory is given in Table 3. For Algorithm 3, the AI is high for DRAM and L3. The AI in L1 is the same in both cases, which is expected since the fractional interpolation is carried out in the same way for Algorithms 2&3.

The second part of the beamformer operation is the weighting and addition of the sensor data realized using an optimized add function. Table 4 presents a comparison of this final step in the beamformer for Algorithm 2&3, for different data sizes. In Algorithm 2, the additions for a beam direction take place in the inner loop, and hence it is cached, making it faster. Hence, for Algorithm 2, the memory bandwidth is very low for DRAM and L3, and small data size for L2 as well (Table 4). For Algorithm 3, the beam loop is inside the sensor loop and beam data needs to be fetched each time (lines 21, 32, 45). To enable the reuse of data, a temporary buffer is allocated

Table 5. Arithmetic intensity for sensor data addition with integer delay operation in A2 & A3

Data size	Arithmetic Intensity (Sensor Addition)							
	L1		L2		L3		DRAM	
	A2	A3	A2	A3	A2	A3	A2	A3
1	0.17	0.17	29.6	0.27	1.3e3	1.1e3	5.7e4	3.1e3
2	0.17	0.17	0.67	0.26	3.8e3	1.0e3	2.4e5	1.3e3
4	0.17	0.17	0.21	0.21	4.8e5	43.09	3.0e5	917
6	0.17	0.17	0.19	0.18	4.9e5	0.532	3.3e5	776
8	0.17	0.17	0.16	0.17	6.1e5	0.355	3.1e5	945
10	0.17	0.17	0.17	0.17	6.7e5	0.308	3.3e5	386

for the beam output of each core (line 8). A final output copy to DRAM is done (line 52). In Table 4 the bandwidth for L1 and L2 is high for smaller data sizes (Table 4), indicating that the buffer is cached in L2. As the data size increases, the buffer gets cached in L3. The DRAM bandwidth is negligible as seen in Table 4.The AI of the weighting and addition operation in the beamformer is given in Table 5. AI values for Algorithm 3 are high for L3 and DRAM, indicating better reuse memory. But these values are smaller than those of Algorithm 2.

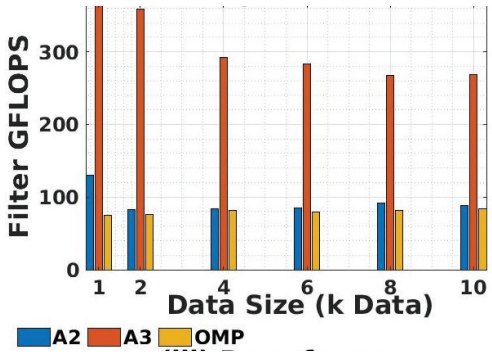
The overall impact of Algorithm 3 over 2 is that the memory transfers from DRAM, as well as L3, are reduced for operations of the beamformer, thereby increasing AI. This is reflected in Tables 2-5.

4.1 Comparison with OpenMP Standard Implementation

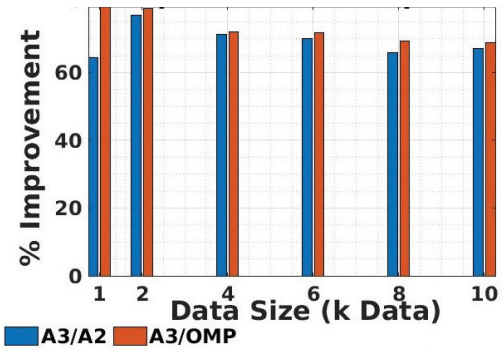
Table 6 provides a comprehensive summary of the impact of Algorithm 3 over 2, as well as Open MP standard-based implementation. This is tabulated for interpolation filter operation and the overall beamformer. Fig.4i shows the performance of the interpolation filter, in which Algorithm 3 outperforms Algorithm 2 and OpenMP by about 2.5-3 times in all cases (200-250 GFLOPS better). Fig. 4(b) displays the percentage improvement factor on an average of 69 % and

Table 6. Performance improvement of GFLOPS for interpolation and beamformer operation [A3 over A2 & OMP]

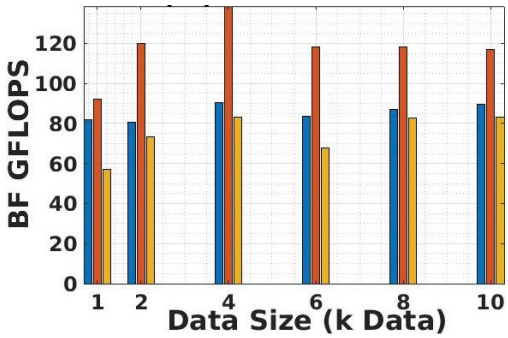
Data size (kData)	Percentage improvement of GFLOPS			
	Interpolation		Beamformer	
	A3/A2	A3/OMP	A3/A2	A3/OMP
1	64.3	79.5	11.2	38.1
2	76.9	78.8	32.6	38.9
4	71.3	72.1	34.7	40.0
6	70.1	71.9	29.2	42.6
8	69.8	69.3	26.4	30.2
10	67.1	68.9	23.1	28.7



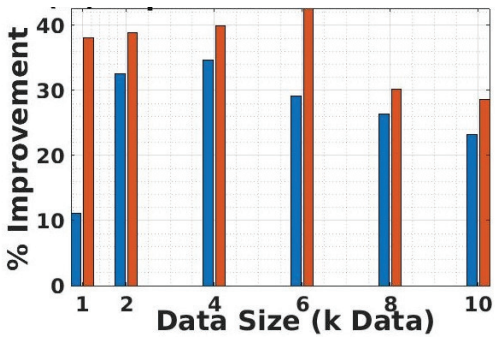
(a) Interpolation filter



(b) Improvement interpolator



(c) Beamformer



(d) Improvement beamformer

Figure 4. GFLOPS achieved for interpolation, (a) Beamformer (b) Beamformer (c) Percentage improvement of Algo 3 over 2 and OpenMP for interpolation (d) for all data sizes.

73 %. The overall performance of the beamformer using Algorithms 2, 3, and OpenMP is shown in Fig.4(c). Algorithm 3 yields higher values in all cases comparatively. Fig. 4(d) As seen from Fig. 4(d) and Table 6, the overall improvement of Algorithm 3 (CABF) is 26 % and 36 % respectively over Algorithm 2 and OpenMP. This demonstrates the overall effectiveness of Algorithm 3. These findings highlight the significance of optimizing computational performance in the context of multi-core processors.

5. CONCLUSION

The article presents the design of a modified CABF algorithm for a DAS beamformer within the context of multi-core processors enabling a cost-effective realization of complex large arrays in Sonars, Radars, medical imaging, and the like. A brief analysis of the beamformer is performed to break down the algorithm, enabling it to use the multicore resources. Modifications are carried out to facilitate the reuse of sensor data from the cache. To characterize, a case study with a 6000 sensor array forming 200 beams is made. To benchmark the performance an Open MP standard-based implementation is made and studied for different data sizes. Results demonstrate that the CABF algorithm reduces memory transactions, as evidenced by the bandwidth numbers. The CABF algorithm has improved FMA calculations by 73% over OpenMP and 69 % over pthreaded implementation. Finally, it is demonstrated that the CABF algorithm yields an overall performance improvement of 36% over Open MP and 26 % over the pthreaded implementation.

REFERENCES

1. Zimmerman, R.; D'Spain, G. & Chadwell, C. Decreasing the radiated acoustic and vibration noise of a mid-size AUV. *IEEE J. Oceanic Eng.*, 2005, 179-187. doi: 10.1109/JOE.2004.836996.
2. Chapman, N. & Price, A. Low-frequency deep ocean ambient noise trend in the northeast pacific ocean. *JASA Exp. Let.*, 2011. doi: 10.1121/1.3567084
3. Miasnikov, E. Can Russian strategic submarines survive at sea? The fundamental limits of passive acoustics, science and global security, pp. 213-251 doi: 10.1080/08929889408426401
4. Dawe, R. Detection threshold modelling explained in DSTO Aeronautical and Maritime Research Laboratory, 1997.
5. Prakash, Narayanan; Gopal, M.G. Vijay & Rajesh, R. Design and implementation considerations for a 3D beamformer on state-of-the-art multicore processors in OCEANS, Chennai, 2022. doi: 10.1109/OCEANSChennai45887.2022.9775501.
6. Urick, R. Principles of underwater sound for engineers. McGraw Hill, 1967.
7. Vlassopoulos, N. & Reisis, D. Conflict-free parallel memory accessing techniques for FFT architectures. *IEEE Tran. on Cir. and Sys.*, 2008, **55**(11). doi : 10.1109/TCSI.2008.924889.
8. Tsai, P.Y. & Lin, C.Y. A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling. *IEEE Tran. VLSI Sys.*, 2011, **19** (12). doi : 10.1109/TVLSI.2010.2077314
9. Cheng, C. & Yu, F. An optimum architecture for continuous-flow parallel bit reversal. *IEEE Signal Proc. Let.*, 2015, **22**(12). doi :10.1109/LSP.2015.2470519
10. Mucci, R. A comparison of efficient beamforming algorithms. *IEEE Tran. on Acou, Speech and Signal Proc.*, 1984 **32**(3), 548–558. doi :10.1109/TASSP.1984.1164359
11. Li, B.; RongY.; Sun, J. & Teo, K.L. A distributionally robust minimum variance beamformer design. *IEEE Signal Proc. Let.,* 2018, **21**(1), 105–109. doi :10.1109/LSP.2017.2773601
12. Synnevag, J.F; Austeng. A; & Holm, S. Adaptive beamforming applied to medical ultrasound imaging. *IEEE Tran. on Ulatrasonics, Ferro, and Freq Con,* 2007 **54**(8), 1606–1613. doi :10.1109/TUFFC.2007.431
13. Amaro. J; Yiu. B.Y.S.; Falcao, G.; Gomes, M. & Yu, A. Software-based high-level synthesis design of FPGA beamformers for synthetic aperture imaging. *IEEE Tran. on Ulatrasonics, Ferro, and Freq Con.,* 2015*,* **62**(5), 862–870. doi :10.1109/TUFFC.2014.006938
14. Kidav, J.U. & Sreejeesh, S.G. An FPGA-accelerated parallel digital beamforming core for medical ultrasound sector imaging. *IEEE Tran. on Ulatrasonics, Ferro, and Freq. Con.,* 2022, **69(**2), 553-564. doi:10.1109/TUFFC.2021.3126578
15. Yeh, C.Y.; Chu, T.C.; Chen, C. & Yang, C. A hardware-scalable dsp architecture for beam selection in mm-wave MU-MIMO Systems. *IEEE Tran. on Cir. and Sys.,* 2018, **65**(11), 3918–3928. doi: 10.1109/TCSI.2018.2856124.
16. Narayanan, M.P.; Gopal, G.V. & Rajesh, R. Accelerating performance of a real-time 2D delay-sum beamformer on general purpose processors. International Symposium on Ocean Technology (SYMPOL), 2021 doi: 10.1109/SYMPOL53555.2021.9689448.
17. Nielsen, R.O. & Boston. Sonar signal processing. Artech House, 1991.
18. Li, Q. Digital sonar design in underwater acoustics - principles and applications. *Springer,* 2011.
19. Dudgeon, D.E. & Mersereau, R.M. Multidimensional digital signal processing. Prentice Hall, 1984.
20. Pridham, R.G. & Mucci, R.A. A novel approach to digital beamforming. *JASA*, 1978, **63**, 425–434. doi:10.1121/1.381733
21. Butenhof, R.D. Programming with POSIX threads. Harlow, UK: Pearson education, 1997.
22. Mattson, T.G.; He, Y. & Koniges, A.E. The OpenMP Common Core., Cambridge: MIT Press, 2019.
23. Open MP, http://www.openmp.org. (Accessed on 07 December 2023).
24. Greg, S.; Richard, B. & Xiaoyun, Y. Multicore image

processing with OpenMP. *IEEE Signal Pro. Mag.,* 2010, 134–138,
doi: 10.1109/MSP.2009.935452.

25. Allen, G.E. & Evans, B.L. Real-time sonar beamforming on workstations using process networks and POSIX threads. *IEEE Tran on Signal Proc.,* 2000*,* **48**(3), 921–926.
doi: 10.1109/78.824694.
26. Bridgman, J.F.; Allen, G.E. & Evans, B.L., Scalable multi-core sonar beamforming with computational process networks. *In* IEEE Conference on Signals, Systems, and Computers, 2010
doi: 10.1109/ACSSC.2010.5757732.
27. Guo, T.H.P.S. Design and implementation of a real-time multi-beam sonar system based on FPGA and DSP, MDPI. *Sensors*, 2021.
doi:10.3390/s20141425
28. George, A.D.; Markwell, J. & Fogarty, R. Real-time sonar beamforming on high performance distributed computers, *Parallel Comput.*, 2000, **26**, 1231–1252
doi:10.1016/S0167-8191(00)00037-5

CONTRIBUTORS

Mr M. Prakash Narayanan obtained his MTech in Electronics Design and Technology from IISc, Bangalore, India and currently working in NPOL, Kochi, India. His research interests include: Design and development of signal processors for sensor processing, deep learning, and the exploitation of multicore processors for signal processing applications.
His current contribution is: Coding algorithm 3 and simulation of algorithms 3,2,1. Evolution of idea behind the paper. Compilation of results and authoring of the paper.

Mr G. Vijay Gopal obtained his MTech in Communication Systems from IIT, Madras and currently working in NPOL, Kochi. His areas of interest include: Signal processing for underwater applications, system design for real-time signal processing, accelerated and embedded computing.
His current contribution is: Coding and simulation of algorithm 1, 2 and its simulation. Expert consultation in tools used, its setup, and the environment of the study. Editing of the paper.

Dr R. Rajesh obtained his PhD in Physics from Jamia Millia Islamia, New Delhi and currently working in NPOL, Kochi. His areas of research include: High-power chemical lasers for directed energy applications and development of fiber optic underwater acoustic sensors.
His current contribution is: Supervisorial contributions. Editing of paper and its structural modifications discussions and ideas to evolve the work as a paper.