

An Application of GEDAE to Adaptive Beam Forming

M. Roadnight
MIEEE
BAE SYSTEMS,
Under Water Systems Division,
Elettra Avenue,
Waterlooville,
Hampshire PO7 7XS
(023) 9226 4466

martin.roadnight@baesystems.com

ABSTRACT

The paper describes the use of GEDAE to implement an adaptive beamformer on embedded hardware. It describes some of the problems encountered, and some of the benefits of using GEDAE.

1. INTRODUCTION

This paper describes the experience of using GEDAE to implement Nunn's frequency domain adaptive beam forming algorithm[3].

To some extent GEDAE automates the process of coding embedded hardware. It eliminates the need to manually allocate buffers and program data transfers etc, but it does not necessarily free the programmer from considering the hardware resources.

In the case when an algorithm may be easily mapped into parallel processes with little inter-communication, the embedded design may be produced with little consideration of the hardware. Indeed, it may be better to produce a more hardware independent design and, if necessary, compensate for any inefficiency by increasing the number of processors.

If this is not an option, e.g. when data distribution is the limiting factor, the performance can only be improved by using the hardware more efficiently. This can be done by: -

Reducing the number of transfers or by making them more efficient.

Adjusting the size of the variables to make best use of the fast access local memory.

Adaptive beam forming provides a good illustration of these points, because it can be demanding in both processing and communications aspects.

For those unfamiliar with the subject, the beams are formed by summing the outputs of a sensor array. To steer the beam in the required direction, it is necessary to compensate for the differences in the propagation delay from the source to each of the

sensors, so that the signals are summed coherently. In the frequency domain, this can be done by phase shifting the sensors' spectra, i.e. by multiplying them by complex coefficients, known as 'weights'.

Adaptive beam formers attempt to optimise the output by steering nulls in the beam response towards sources of interference. Often this is done by maximising the signal to noise ratio. In the well-known minimum variance algorithm (reference [1] and [2]), the beam former weights are adjusted to minimise the output power, subject to constraints that control the shape of the beam in the desired direction, i.e. the 'look' direction.

2. ADAPTIVE FREQUENCY DOMAIN BEAM FORMING

The beam power output, P, is given by: -

$$P = W_k^T Z_k \quad \text{Equation 1}$$

Where Z_k is the vector of M sensor inputs for the kth frequency bin, and W_{kopt} is the optimum weight vector, given by: -

$$W_{opt} \propto [Q^{-1}C] \quad \text{Equation 2}$$

where C is the constraint vector and Q is an M x M matrix obtained from the spectral densities: -

$$Q = \sum_k Z_k^* Z_k^T \quad \text{Equation 3}$$

Where K is the total number of frequency bins in the band.

To reduce the number of freely adaptable weights, Nunn[3] introduces a pre-conditioning matrix, B: -

$$B = \left[\left| I_{JM} \right| \cdot \left| \frac{(k - km)}{km} I_{JM} \right| \cdot \left| \frac{(k - km)^{J-1}}{km} I_{JM} \right| \right]$$

where, I_{JM} is the $JM \times JM$ identity matrix and ‘J’ is a parameter that defines the order of the polynomial. Typically, its value is between 1 and 3.

This allows the weights to be defined in terms of a ‘super weight’ vector.

The weights are given by: -

$$W(k) = B(k)^T V \quad \text{Equation 4}$$

where V is the ‘super weight’ vector. This leads to the solution: -

$$V_{opt} = Q^{-1} C'^* [C'^T Q^{-1} C'^*]^{-1} F^T \quad \text{Equation 5}$$

where ‘*’ denotes the complex conjugate and ‘T’ denotes the transpose, and,

$$Q = \sum_k B_k^H Z_k^* Z_k B \quad \text{Equation 6}$$

and

$$C' = B^T C \quad \text{Equation 7}$$

C is a matrix defining the constraints, e.g. the steer direction.

Therefore, the algorithm requires the following operations (see Fig 2): -

1. Calculate an FFT for each sensor input.
2. Calculate the correlation matrix, Q.
3. Invert Q.
4. Calculate the optimum ‘Super Weight’, V_{opt} .
5. Calculate W_{opt} .
6. Form each beam (equation 1).

Note that step 6 is a frequency domain convolution, and that the FFTs in step 1 have to be overlapped to make the process continuous.

3. IMPLEMENTATION WITH GEDAE

The design was implemented in two stages. First a graph was constructed to run on a NT/Unix host, and then this was modified to run on the embedded hardware.

The Host graph used primitives from the GEADE library, wherever possible, so the beamformer was relatively easy to construct. To prove its correct functional operation, the model also had to include signal generation and a GUI to provide a means of plotting the beam and controlling the beamformer parameters, e.g. steer direction.

An embedded design was produced by completing GEDAE’s partition table, but, the resulting trace diagram showed that the partitioning had to be optimised to prevent processes stalling because of delays in the data transfers.

GEDAE’s trace facility was used to benchmark some of the functions, e.g. matrix inversion, on the hardware. Estimates were also made of the memory required using the schedules produced by GEDAE.

The final embedded design embedded design was different to the Host model, and took as long to complete.

A number of new ‘eval’, ‘apply’ and ‘trigger’ primitives were required. The Host model used contained 71 primitives, of which 47 were from the libraries. Six custom primitives were required for plotting the beam, i.e. to generate the signal and beamformer steer angles and synchronise the increments in their values to the beam outputs block length. The beam former required a further 12 primitives to generate the constraint matrix and the polynomial matrix, B(k), etc. Four new primitives were required in the embedded design to re-arrange the data for transfer between processors (see option 2, below).

3.1. Benchmarking

Table 1 describes the hardware used in the benchmarking.

Table 1

Vendor	Sky	Mercury
CPU	MC7400	MC7400
Clock Speed	333 MHz	400 MHz
L1 cache	32 K bytes	32 K bytes
L2 cache speed	333 MHz	160 MHz
External Memory	83.3 MHz	133 MHz
Communications	Sky Channel 320 Mbytes/s	Raceway 66 MHz 267 Mbytes/s

Figure 1 shows the variation in the time to calculate an FFT butterfly versus the size of the FFT for Sky and Mercury hardware. This increases as the FFT gets smaller because of the overheads in calling and setting up the routine. The performance falls again for FFTs above 2K, because the data exceeds the L1 cache.

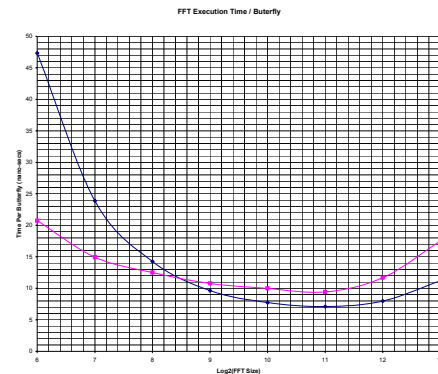


Figure 1 FFT Execution Time / Butterfly

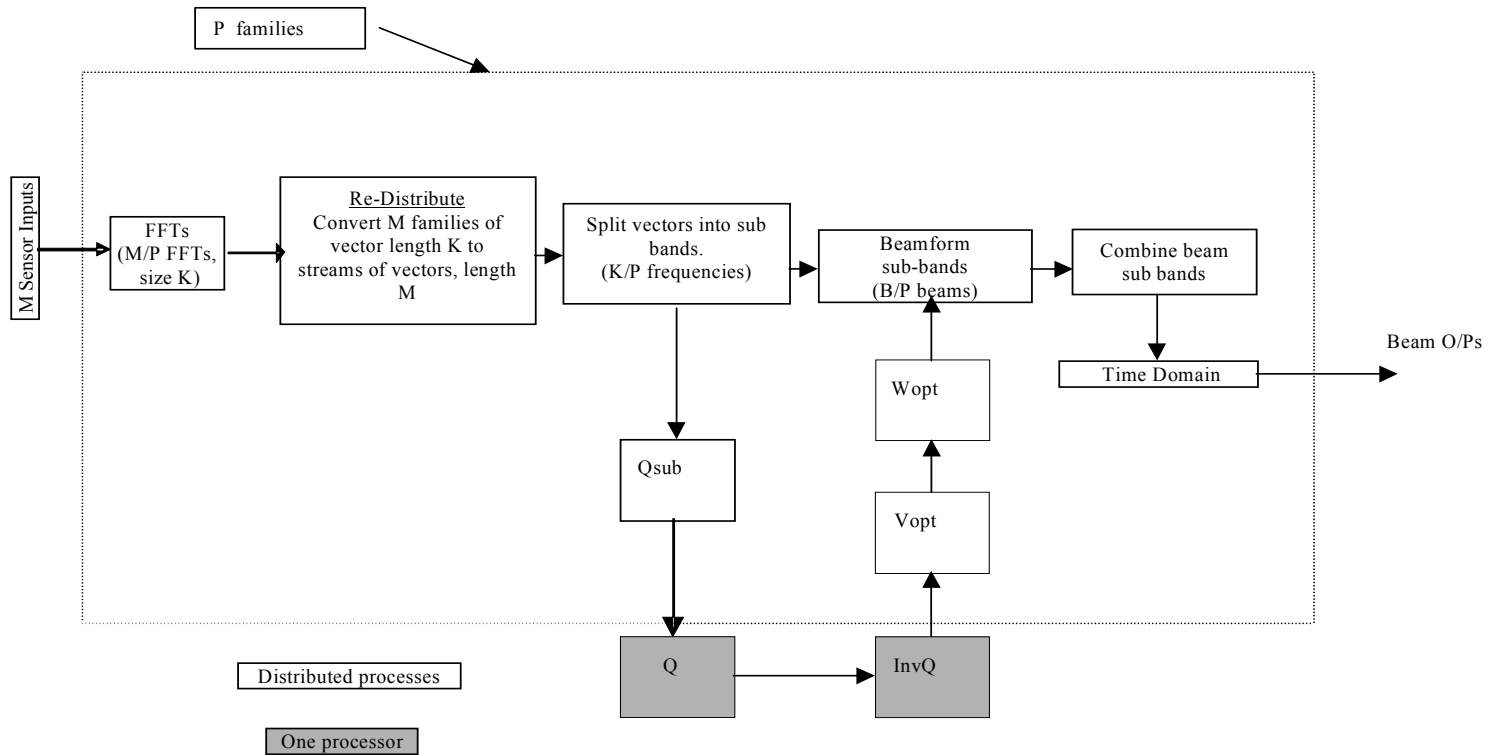


Figure 2 Frequency Domain Beam Forming Operations

The inversion of the correlation matrix is done using the GEDAE “mx_inv” library function, which uses LU decomposition. This function dominates the processing for large matrixes, because the number of calculations required increases non-linearly with the size of the matrix, as shown in Figure 3.

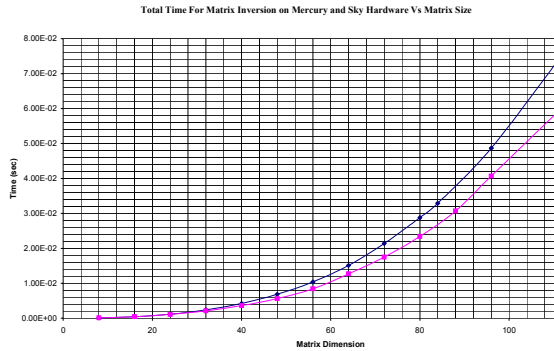


Figure 3 Total Time for Matrix Inversion on Mercury and Sky Hardware Vs Matrix Size

It has been noted that the Mercury ‘SAL’ library includes a matrix inversion routine, ‘cmatludx’, that is not used by the GEADE primitive. The execution time measured for a 32 element matrix is approximately three times longer than the time quoted for ‘cmatludx’.

BHDS suggest that the mx_inv primitive could be modified to call the SAL library function.

3.2. Calculation of Correlation Matrix

This requires the summation of vector cross products, which can be implemented using the vx_conj, vx_crspr and mxn_sum primitives from the GEDAE library.

The input consists of a stream of K vectors of size JM complex words, where M is the size of the sensor array (M), ‘J’ is a parameter (see the formulae above) and K is the number of frequency cells. The input to the cross product is 8JMK bytes but its output is $8(JM)^2K$ bytes.

An attempt was made to reduce this to $8M^2$ by introducing a sub-schedule around the conjugation and dot product primitives, but this failed because the mxn_sum primitive decimates by K. This is a static primitive, and therefore requires all K input matrixes.

This could be overcome by making the schedule dynamic. A more efficient solution is to calculate the cross product and the summation in one operation, i.e. to replace the vx_conj, vx_crspr and mxn_sum primitives with one custom coded ‘static’ primitive. This new primitive has a granularity of K, and its input buffer is reduced to 8MJK bytes, i.e. by a factor of MJ. Alternatively, a cyclic primitive could be used, as suggested by BHDS.

3.3. Partitioning

The aim is to distribute the processing load without exceeding the fast access memory available to each processor, while keeping the delays caused by data transfers acceptably low. These are conflicting requirements!

Depending on the size of the array and the matrix ‘B’, the calculation of the correlation matrix may be a significant part of the processing and the memory requirement. Moreover, its calculation requires all the sensor inputs and all the frequency components, so cannot easily be partitioned for calculation across multiple processors, without requiring the transfer of large amounts of data between them.

The processing load depends upon the rate at which the inverse is calculated, i.e. the rate at which the optimum weight vector, “Vopt”, is updated. This could be reduced either by increasing the length of the FFTs, or by averaging the correlation matrixes of multiple FFT frames. However, the weights are only optimized for the data block used to calculate the correlation matrix, so the FFT outputs must be saved until the beam forming can be calculated. Therefore reducing the weight update rate increases the memory required.

Two methods of partitioning were considered: -

1. Time multiplex the task, so that one processor calculates all of the beams, but in the time taken to collect the data for multiple FFTs frames.
2. Partitioning the processing into sub-bands.

The first option is the simplest. It effectively divides the processing load between the number of processors, but it also increases that the delay in calculating the output. To compensate, the length of the data block used to calculate the correlation matrix may reduced, but this increases the rate at which the matrix inversion fires.

Option 2 has less delay and requires less memory, but the graph is more complicated and there is a greater overhead in transferring the data between processors. Also, we need to distribute the matrix inversion, to get the maximum benefit from increasing the number of processors. This requires a new primitive, and in any case, cannot be done using Cholesky’s method.

The choice between these methods may depend on the hardware selected, the maximum weight update period, the number of sensors, and hence the size of the schedule and the memory available. With so many variables, some experimentation is needed to find the best compromise. The schedules required are complex, especially for option 2, but fortunately, GEDAE generates these and provides information on the memory usage and the timing of the processes and data transfers.

Both the Sky and Mercury hardware have 2Mbytes of L2 cache, plus external memory, so storing intermediate results is not a problem.

Communication by Raceway or SkyChannel is inefficient for transferring small data blocks, because of the overheads in routing the messages. Moreover, it was found that broadcasting data to multiple processors was implemented as separate writes to each processor. It is therefore better, in this case, to avoid sharing large amounts of data between processors. When this is necessary, the efficiency of the transfer may be improved by partitioning the processing so that the data is sent in large blocks, or by re-packaging the data streams e.g. by sending scalars as vectors, or converting families of vectors to matrixes.

Option 1 tends to be more suited to the Mercury and Sky hardware, because, although it requires more memory, it needs less I/O bandwidth than option 2. However, the optimum graph may well be different with the TigerSharc [5], which has less fast access memory, but supports multiple connections via its link ports and allows broadcast writes between processors, within the same cluster of processors.

4. PERFORMANCE

Table 2 gives the parameters of the beamformer used in this example.

Table 2 Beam former Parameters

No. of Sensors	FFT Size	Overlap	Useful O/P	No. of FFTs	J
49	256	39	217	1	1

Table 3 Performance on Mercury Hardware

Parameters				
Sensors	49			
FFT size (Nbf)	256			
Overlap (R)	39			
Number FFT frames over which correlation matrix inverse is calculated (Average)	1			
Measurements On Mercury Hardware				
Function	2 Processors		4 Processors	
	Time (ms)	Percent	Time (ms)	Percent
FFTs and read data	1.5	6.8%	0.8 ms	4.8%
Re-package	0.9	4.1%	0.2 ms	1.2%
Re- distribute data	0.9	4.1%	0.1 ms	0.6%
Correlation Matrix (Q)	6.7 ms	30.5%	2.3 ms	13.9%
Invert Q	6.3 ms	28.6%	7.3 ms	44.2%
Calculate Weights	0.2 ms	0.9%	0.2 ms	1.2%
Beam Outputs	6.3	28.6%	2.3	13.9%
Convert to Time Domain	0.072	0.3%	0.05	0.3%
Stalled	-	-	2.9 ms	17.5%
Total	22 ms		16.5 ms	
Maximum sample rate	10 kHz		13 kHz	

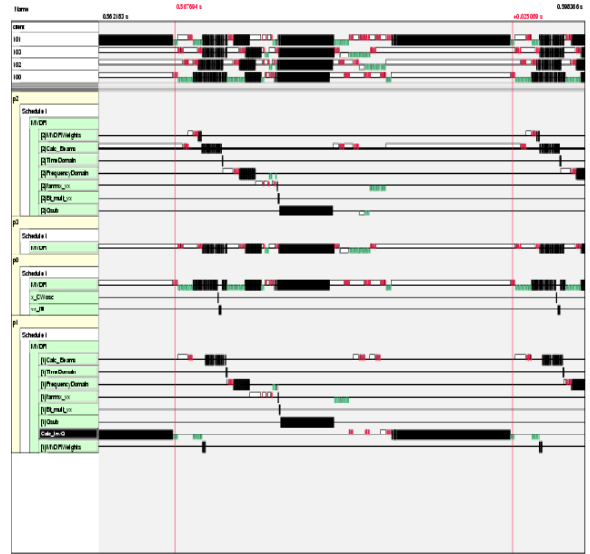


Figure 4 Trace diagram for Sky Hardware

When run on a single PowerPC on the Mercury hardware, the processing takes approximately 31.5 ms. Reading the input data takes approximately 2.5 ms, making a total of 34ms. This would allow a maximum input sample rate of 6.3 kHz.

Extrapolating from this suggests that option 1 may operate at up to 12 kHz with two processors and 24 kHz with four.

Table 3 summarises the performance of option 2 with two and four processors. The total time to distribute the data with four processors was predicted to be less than 2ms. The measured time is approximately 1 ms, but the processors stall for 2.9 ms, presumably because of the schedule generated by GEDAE.

Figure 4 shows the trace table for the beamformer using four processors on the Sky hardware. This used 'stream shared memory' transfers, so the total processing time was longer than on the Mercury hardware (approximately 24 ms). It is expected that there would be little difference with more optimised data transfers on the Sky hardware.

A better performance can be expected, if the weight update period is increased, either by increasing the size of the FFT, or by averaging the correlation matrix over FFT frames. With a 2k FFT, the processing takes approximately 30 ms, but the processors are idle for 5ms i.e. for approximately 20% of the time. The maximum sampling rate increases to 18 kHz.

The data transfers should ideally be handled by DMA and, ideally, these should be concurrent with the processing. So far, the schedule produced by GEDAE fails to achieve this, causing the processors to stall. The reason for this is not clear, as the data

transfers should only require 5% of the total time taken to process the data.

Table 4 Option 2 Memory Requirement Vs Beam Former Parameters

Nbf	FFT Frames	J	Memory (Mbytes)	
			P0/P2/P3	P1
256	1	1	0.215	0.236
256	10	1	2.11	2.326
2048	1	1	1.544	1.544
256	1	2	0.226	0.572
256	1	3	0.361	1.139

Table 4 shows the variation in memory with the beam former parameters, using option 2. Note that the TigerSharc’s internal memory is only 6Mbits, i.e. 0.75 Mbytes.

Migrating the graph from Sky to Mercury hardware took approximately 1-2 weeks. It is intended to repeat this with the TigerSharc when it becomes available.

5. CONCLUSIONS

GEDAE provides a lot of help in the design of embedded software for multiple processor systems. The use of families and route boxes, allow the graph to be drawn for a variable number of processors. It generates the schedule and provides information on the memory use and the timing of the processing. This facilitates

changes to the number of processors, the allocation of processes etc., to achieve the best performance. However, as illustrated above, it may require some experience and experimentation to find the best way to implement the schematic.

The memory required for the calculation of the correlation matrix has been significantly reduced over that initially required using the library functions. Likewise, data transfers have been made more efficient by increasing the size of the packets sent between processors.

Depending on the size of the sensor array and the ‘J’ parameter, the execution time may be dominated by the matrix inversion. It may be possible to reduce this by calling of the vendor’s matrix inversion routine.

In this example, the time-multiplexed graph is the easiest to implement, since it requires fewer custom coded primitives. Option2, partitioning by sub-bands, requires further work to get the best performance from the hardware, both in the distribution of the data and the calculation of the matrix inverse.

6. REFERENCES

- [1] Hudson, J. E., ‘Adaptive Array Principles’, Peter Pergrinus for IEE, 1981. ISBN 0-906048-55-9
- [2] Monzingo, Robert A., and Miller, Thomas W., .Introduction to adaptive arrays, Wiley 1980, ISBN-0471057444
- [3] Nunn, D., Sub-optimal Frequency-domain Adaptive Antenna Processing Algorithm For Broadband Environments, IEE Proceedings, July 1987.
- [4] O.L. Frost. “An Algorithm for Linearly Constrained Adaptive Processing”, IEEE Proceedings, August 1972.
- [5] Tigersharc Hardware Specification, Analog Devices.