

Investigating the efficiency of a GEDAE graph

Simon Challands

Thales Underwater Systems
Dolphin House, Ashurst Drive
Stockport, Cheshire SK3 OXB, UK
+44 (0)161 741 3982

simon.challands@uk.thalesgroup.com

ABSTRACT

We describe the results of a comparison between the processing load taken by a GEDAE graph and estimates of the loading that could be produced by hand-written code, using a typical sonar processing component (a beamformer). The results show that there is some processing overhead from developing in GEDAE, the beamformer example running at 125% of the estimated loading. Both the methods of timings, and the areas where we found the greatest discrepancies are discussed, and suggestions for improving the performance in high-load parts of the graph are put forward.

1. Introduction

This paper investigates the processing efficiency of GEDAE boxes compared to estimates for handwritten code, and examines some methods for improving a graph's performance. The measurements and estimates are made for a typical sonar processing component (a beamformer). The beamformer used is not one used in any particular project, and the figures chosen for number of beams, data rates etc. are arbitrary.

2. The Beamformer

The principal components of the beamformer are shown below in Figure 1. The numbers in Figure 1 were chosen as being representative of a hypothetical system, but are not taken from any real system (the non acoustic rate of 16Hz was chosen simply because it gives a simple relationship between the acoustic and non acoustic rates, for example).

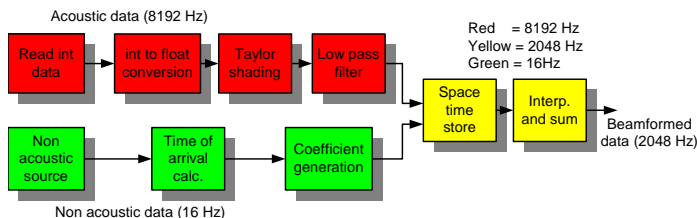


Figure 1: Beamformer diagram

The acoustic data is reading 64 hydrophones per update, but the beamformer is a half aperture beamformer, and just selects the central 32 (from Taylor shading onwards). The output is 64 beams wide.

Acoustic data is read from file, then converted into floats for easier processing. It is low pass filtered to prevent any aliasing effects in the output.

3. GEDAE graph Performance

Any overheads involved in using GEDAE need to be measured. Timing various GEDAE boxes, or the overall graph execution alone cannot do this because there is no way of separating out the times. An estimate of the expected execution times can be produced, and anything measured on the GEDAE graph compared with these numbers.

Any results obtained of GEDAE overheads will by nature be tenuous, as they will greatly depend on how the graph was implemented. Clearly a graph consisting of a few boxes containing large amounts of complex in-house code will have proportionally fewer GEDAE overheads than one that consists of large numbers of simple boxes. Also, more time spent optimising GEDAE parameters (e.g. granularity multipliers) can potentially have a significant impact in the performance. Very little effort has been made here to investigate this for the beamformer, other than to notice that a large increase in granularity will not always improve the performance. This is most likely due to the greater memory requirements of large granularity resulting in the OS having to swap in and out memory pages more often.

3.1 Measurement Methods

Measurements can be made in three ways. The first two involve using the trace table, and with this tool individual box timings can be made, or a larger block of data can be measured.

Using the trace table has the advantage of seeing where there are large discrepancies from the predicted results, but gathering the trace table measurements imposes an additional overhead on the processing (which will be greatest on graphs consisting of a large number of fast-firing boxes).

An alternative method of using the trace table to measure performance is to measure the time taken to process a large number of firings (e.g. over the whole time the trace table statistics were being gathered). Information can be obtained from it saying how many firings there were within that time, so an average can be calculated.



The third method involves turning the trace table off, thus removing any overhead from it, and adding timing code to another box. The read in box was used for this purpose. It counts how much data it has read, and measures how long it takes to read in ten seconds' worth of data. The reading cannot be made quicker than the data can be processed, so this measures the overall graph performance (but obviously has its own overhead).

4. Predicted timings

The predicted results were made by calculating the product of the number of multiply / accumulates or other operations used by each process, and multiplying them by the number of times each process is used. These timings are presented in Table 1. The columns in Table 1 show the following:

Operation: The operation on which the estimate is based.

Actual operation: The operation performed by the GEDAE graph

Cost: A weighting for additional functionality not covered by treating the operation as a collection of a single instruction (e.g. loop counter overheads).

Fir: Equivalent to the box's decimation rate

Npts: Number of points in the low pass filter

Beam: Number of beams processed at this point

Staves: Number of samples processed each time (this does **not** include GEDAE's granularity).

F. Calc: The number of times the calculation needs to be made in a second.

Mops / s: How many millions of processor operations are required per second

Operat. The type of operation. Each operation has had its average number of processor cycles estimated. The operations are i/o for input / output (actually estimated for an Ethernet i/o, so only provided for reference), fixflot for int to float conversion, maxr for multiply / accumulate, and rwExt for memory access outside of the processor cache.

Theor: Theoretical number of millions of processor cycles required each second for each operation, simply calculated by multiplying the previous columns.

total: Estimate of how many millions of cycles per second are required for the whole beamformer.

+marg: ~20% margin of error

Operation	Actual operation	Cost	Fir	Npts	Beams	Staves	F.Calc	Mop/s	Pentium 2.8GHz			
							(Hz)		Mcy/s	Theor	total	+marg
Input	Read from file					64	8192	0.524		25.17		
Float	Si to S box					64	8192	0.524		2.10		
Aperture weighting	Aperture weighting					32	8192	0.262		1.05		
Low pass filter	Low pass filter			111		32	8192	29.1		116.39		
Coeff gen	Time of arrival	3			64	32	16	0.098		0.39		
Coeff addressing	Shift and coeff	6			64	32	16	0.197		0.79		
Data Accessing	Vspace time store		2		64	32	2048	8.389		58.72		
2pt BFM	Interpolation & sum		4		64	32	2048	16.78		67.11		
Output					64		2048	0.131		6.29	278	333.6

Table 1. Predicted timings

4.1 Results

The comparison between predicted and measured times is shown in Table 2.

The graph's granularity multiplier was set to 4. This produced a measurable speed increase over leaving it at 1. Increasing the granularity beyond this did not appear to result in an significant gain in performance, and increasing it very much higher resulted in worse performance (see section 6).

Table 2 lists the comparison between predicted and measured times for the various operations in the beamformer (see Figure 1). There are still additional factors that are not taken into account in those measurements, namely the trace table overheads and background process time. Section 5 attempts to determine the impact of the trace table on these measurements. Time spent by the processor on background (e.g. OS housekeeping) tasks cannot be separated from the other timings by the methods used here, but is expected to be small.

The "upsampling" measured result in the "Others" row is where the graph uses the 16Hz non acoustic data with the 8192 Hz acoustic data. GEDAE is repeatedly copying the non acoustic data to upsample it, instead of just reusing the same values several times

over. There is no simple way around this problem without vastly increasing the granularity of the spacetime store, which has a severe detrimental effect on its performance. Recoding the spacetime store to use non deterministic dataflow for the acoustic data (which means it will only take data from that chain when it is available) may be a solution to this problem, but this has not yet been investigated, and would involve some graph redesign.

In measuring the processing load there was no data output at the end, which is why the final estimate figure is slightly lower than the one shown in Table 1.

The results shown in Table 2 fall into two groups, processes that are made up largely of very simple GEDAE primitives, and those that contain more complicated hand-coded primitives. Those processes belonging to the former category are the int to float, weighting, shift and coefficient calculation, and interpolation and sum, and these are also those parts of the graph that run slowest in comparison to their predicted times. The simpler the box is the less it has to do in the Apply method, so in the case of a very simple box most of the code is in the box's interfaces. A box with a complicated Apply method does not have fewer overheads, but the percentage of its total execution time spent on these overheads is far less.

Operation	Frequency of operation	Amount of data processed	Predicted time (10 ⁶ cycles / s)	Measured time (10 ⁶ cycles / s)	Difference
Data in*	8192 Hz	64 hphones	25.17	19.23	76%
int to float	8192 Hz	64 hphones	2.10	3.89	185%
Weighting	8192 Hz	32 hphones	1.05	3.72	355%
Low pass filter	8192 Hz	32 hphones, 111 coeffs.	116.39	130.40	112%
Time of arrival	16 Hz	64 beams, 32 hphones	0.39	0.36	92%
Shift and coeff. calc.	16 Hz	64 beams, 32 hphones	0.79	2.07	263%
Spacetime store	2048 Hz	64 beams, 32 hphones	58.72	66.52	113%
Interpolation and sum	2048 Hz	64 beams, 32 hphones	67.11	150.87	225%
Others	-	-	54.34 (20% error estimate) Plus unknown OS overhead	51.28 (upsamp. etc)	113%
TOTAL			326.06	428.34	131.37%

* Predicted load is for Gigabit Ethernet, whereas in the GEDAE graph data was being read from disc, so these figures are not readily comparable.

Table 2. Predicted vs. Measured Timings on Individual Operations

5. Trace Table Overheads

The first of these is the overhead in the GEDAE model from gathering the statistics. Normally GEDAE would be run with this feature disabled, so it should not be considered part of the processing loading. GEDAE Inc. states that the times spent gathering measurements will appear on the trace table as times when nothing appears to be done, and will not be included in the box time measurements, so the trace table time will not be included in the numbers shown in Table 2.

In order to determine the impact this has, and to get a comparison outside of the trace table, timing code was added to the read box. This simply notes the time at the start of execution, counts how long it takes to read ten seconds' worth of data, and then subtracts the current time from the start time. Data is being read and processed as fast as possible, so the ratio of actual time taken to ten seconds will be the amount of available processing power used if the input was constrained to be read at the actual sample rate. Table 3 shows the impact of gathering trace table measurements on this particular application. These measurements do, of course, also add to the loading, so do not measure the entire trace table overhead, but by confining them to one box (the read box) the overheads are eliminated from the rest of the graph. The trace table measured performance of the timed read box is actually the one shown in Table 2.

Comparing the numbers in Table 3 with the final measured result in Table 2 shows that the two ways of measuring times produce comparable results. Using the trace table to measure the time of a whole block of data will naturally include its overheads, and that result matches the "with trace table" row in Table 3.

Mode	Time to process 10 seconds of data	Percentage processor loading
With trace table	1.61 seconds	16.1% (450.8 Mcyc/s)
Without trace table	1.54 seconds	15.4% (431.2 Mcyc/s)

Table 3: Trace Table Loading

These comparisons should not be taken to arrive at a percentage increase for any given graph. The time taken gathering trace table measurements will depend on the number of boxes and the granularity they are firing at. The trace table therefore has an overhead of 4.3% on this particular example.

Removing this trace table overhead from the final total shown in Table 2 gives the final figure of the GEDAE graph running at 125% of the estimated loading.

6. Effects of granularity

Changing the granularity of the graph can have unexpected effects on the performance. Whilst it reduces the overheads spent communicating between boxes it requires more data to be passed between each box, and when increased too high it can have a detrimental effect on performance, as the operating system has to page memory in and out more frequently. This is illustrated in Figure 2, which shows the changes in the mean time taken to process a fixed amount of data. The minimum time is slightly higher than that shown in Table 3 due to the more frequent printing of timings that was being done here.

The fluctuations around the minimum point were most likely due to the trade-off between increases in performance due to lower

inter-box overheads being offset by more memory paging varying as it applied to different boxes.

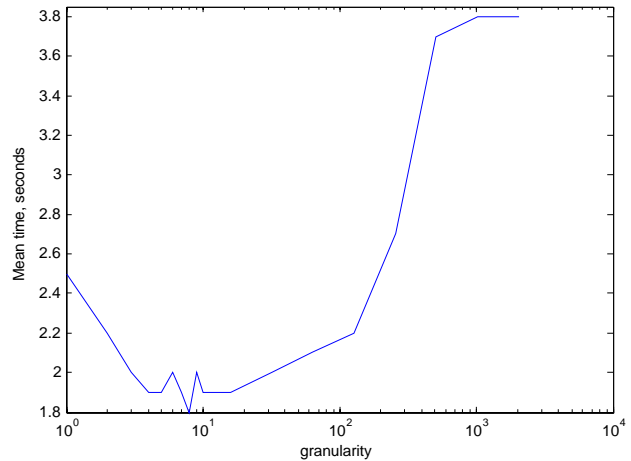


Figure 2. Effect of granularity on graph efficiency

7. Other Overheads

The predicted timings estimate the time spent on actual processing **only**. There are numerous other things going on which will add an additional overhead. Examples are:

1. Procedure calls
2. Ancillary code in functions (e.g. incrementing array indices)
3. Background operating system tasks

The source from which the estimates were taken says that 30% of processing time should be left free for operating system tasks (which could include such things as function calls). Such overheads will naturally be included in the timings in Table 2. Unfortunately there is no simple way of divorcing them from GEDAE-imposed overheads. It is clear that the overheads do not form 30% of the entire processing time, since the complete start-to-end measured load, including processing and any other overheads, would be less than 30% if the data was coming in in real time. It has therefore not been possible to factor in this loading component.

8. Improving Performance

The numbers shown in Table 2 and Table 3 do not represent the performance of the graph as when it was first put together. A few areas stood out as being highly inefficient when compared to the predicted results.

Improved performance, bringing the measured values closer to the predicted ones, was achieved two ways:

1. Improving the code
2. Changing graph parameters and boxes

The first, non-GEDAE specific task involves removing what are essentially non-processing inefficiencies. Certain processes, such as the time of arrival calculation, were non-optimal, and could be re-written by removing parts of calculations from inner loops, for example, so the "functional" parts of the code (i.e. multiply / accumulates etc.) form a greater percentage of the overall number of instructions executed by the processor, but with the penalty of possibly making the code less legible. This is clearly an

improvement that could be made to any application, so in a way is irrelevant to this discussion.

Changing the graph involved replacing hand-coded boxes with built-in GEDAE primitives (or vice versa), and changing the granularity setting. The two changes are interlinked, and probably explain most of the GEDAE overheads.

Using small built-in GEDAE boxes imposes the additional load of an interface into what can be a very simple operation, so there is the potential for them to be quite inefficient, even if their core code is very efficient. This can be significantly reduced by increasing the box's granularity, so that the inside of the box is processing more data each time, thus reducing the interface overhead (and on a multiprocessor system the amount of time spent on communications). The downside of this is that it requires more memory (as it is processing more data in a batch), which can reduce performance if the OS is having to constantly swap in and out memory pages. In this application only a very crude effort was made at investigating the change of performance with the overall graph granularity multiplier. These changes made almost no difference to performance of the relatively complicated hand-coded boxes (e.g. the time of arrival calculations), but had a noticeable effect on the parts of the graph built up of very simple GEDAE primitives (the coefficient generation and the interpolation and sum parts).

The `v_efird` box that was used to perform the low pass filtering was supplied by GEDAE (but is not currently part of the library), but was manually altered to improve performance by removing calls to the `e_function` used to do the multiply-accumulate, and by unrolling the loop over hydrophones. This changed its performance from over 250% of the predicted time to 112%.

9. Conclusions

The final, complete graph is less efficient than predicted, running at about 125% of what is expected. How much this is due to GEDAE and how much it is due to the implementation cannot be completely determined.

It is still possible to draw some conclusions about GEDAE graph components by looking at the individual process performances, shown in Table 2.

Boxes containing more than a few simple instructions operate with very little overhead. The two clearest candidates for this are the time of arrival calculations and the space time store. Remember the individual predicted components in Table 2 do not include the 20% allowance for underestimating. If this is taken into account then these compare very well with the predictions.

Large collections of simple boxes may operate at 200% of the predicted speed, most probably due to interface overheads. These boxes do, however, seem to perform much better at high granularities than hand-coded boxes (GEDAE seems to be managing the memory usage in them well so that they do not cause page swaps as often).

The summary is that a GEDAE graph can produce reasonable performance, if care is taken. Speed-critical parts of the graph should be brought together into a single primitive where possible, with care being taken on the memory usage of the inputs and outputs. Less critical parts can adequately be made of simple library primitives.