Efficient computation of passive seismic interferometry

Jan Thorbecke, and Guy Drijkoningen

Delft University of Technology, Department of Geotechnology, The Netherlands

Abstract

Seismic interferometry is from a computationally point of view based on cross-correlating two signals. One application of seismic interferometry lies in passive seismics. In passive seismic recorders are placed on the surface of the earth and signals are continuously measured. In this abstract we investigated an efficient way of computing the cross-correlation, of a limited number of output samples, for continuously measured signals. Efficiency tests are carried out on two types of common of the shelf 64-bit CPU's: Intel Xeon and AMD Opteron, and IBM's 440d.

Introduction

We have a number of passive measurement stations which are continuously recording (with for example 1 ms sampling in 24 bit) signals from the surface and subsurface of the earth. These recordings are being correlated with each-other to simulate seismic reflection data. In this abstract we mainly concentrate on an efficient implementation of the correlation kernel and how to deal with the data being recorded continuously.
For a good response after the cross-correlation a very long recording time and many spatially uncorrelated white-noise sources are needed. The longer the recorded response, the better it approximates diffuse fields. At the same time, with longer recording times the response from more noise sources is recorded, and as a result, one obtains better illumination of the subsurface. The theory of seismic interferometry has gained a lot of attention in the past 5 years and is for example derived and explained in Draganov et al. (2006); Wapenaar and Fokkema (2006). The basic correlation relation, which is of our interest, is given by:

$$R(\mathbf{x}_A,\mathbf{x}_B,t)+R(\mathbf{x}_A,\mathbf{x}_B,-t)=\delta(\mathbf{x}_{H,B}-\mathbf{x}_{H,A})\delta(t)- T_{obs}(\mathbf{x}_A,t)T_{obs}(\mathbf{x}_B,-t) \quad (1)$$

where $\mathbf{x}_H$ denotes the horizontal coordinates (x_1,x_2). Here $R(\mathbf{x}_A,\mathbf{x}_B,t)$ is the reflection response of an inhomogeneous medium in $\mathcal{D}$, including all internal multiples, for a source at $\mathbf{x}_B=(\mathbf{x}_{H,B},x_{3,0})$ and a receiver at $\mathbf{x}_A=(\mathbf{x}_{H,A},x_{3,0})$. $T_{obs}(\mathbf{x}_A,t)$ is the transmission response of uncorrelated (noise) sources in the subsurface, including all free surface multiples, with receivers at $\mathbf{x}_A$ on the surface of the earth. Equation (1) shows that the correlation of transmitted signals measured at different position at the surface of the earth gives reflection data. The information contained in this reflection data is one of the main goals in passive seismics.

The measured 'noise' signal

The measured signals $a_i(t)$ at position $\mathbf{x}_i$ are assumed to contain transmission events from small earthquakes (caused by layer slides, reservoir depletion) in the subsurface. Besides these transmission events, from deeper parts in the subsurface, the signals will also contain a lot of surface waves caused by 'disturbances' at the surface (traffic, sheeps, people, wind, ...). Our main interest are the transmission events from the deep subsurface. The initiation time of the transmission events is unknown, we assume/hope this will happen a few times a day. By summing over long intervals different events are added together and will enhance the constructed reflection series. We assume that the response of the earth of the top layers we are interested in does not change during the (long) time of the measurements. The events we are interested in will always occur in a short time window (e.g. 10 s.). In Figure 1 a signal with the transmission events occurring at different times t_e^i is shown.

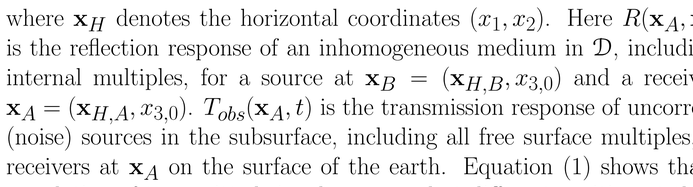


FIGURE 1: *The simplified measured signal contains transmission events which occurs at unpredictable times t_e^i. Each event contains a series of back-reflections from the surface of the earth, and internal reflections between layers within the subsurface.*

Correlation

Given the two real signals $a(t)$ and $b(t)$, we can compute the cross-correlation as:

$$c_{ab}(t)=a(t)*b(-t)=\int_{-\infty}^{\infty}a(\tau)b(\tau+t)\,d\tau \quad \forall \quad t\in(0,T_c) \quad (2)$$

where the correlated signal has a length T_c. This length must be long enough to contain the reflections we are interested in, e.g. 10 seconds. This length is bound by the physical position of the reflectors. Note that cross-correlation is not commutative and $c_{ab}\neq c_{ba}$.

The length of the original input data to be correlated is infinitely long, expressed by the bounds of the integral above. In practice we will have to start somewhere, say at time 0, and we can only correlate up to a certain large time, which we call T_α. Then we have:

$$c_{ab}(t)=\int_0^{T_\alpha}a(\tau)b(\tau+t)\,d\tau \quad \forall \quad t\in(0,T_c) \quad (3)$$

Due to the end points of the total correlation window it is clear that we make an error here.
Our aim is to design a correlation algorithm which continuously computes the correlation result. In this way we can avoid the storage of very long signals. This approach is possible because we are only interested in a short output time-window (T_c). Using the limited number of output samples, the integral above can be written as a finite sum of smaller windows, i.e.:

$$c_{ab}(t)=\sum_{k=0}^{N_w-1}\int_{kT_k}^{(k+1)T_k}a(\tau)b(\tau+t)\,d\tau \quad \forall \quad t\in(0,T_c) \quad (4)$$

where N_w is the number of windows and T_k is the smaller window in which a correlation is calculated.
The discrete version of equation (4) is

$$c_{ab}[i\Delta t]=\sum_{k=0}^{N_w-1}\sum_{j=kN_k}^{(k+1)N_k-1}a[j\Delta t]b[(j+i)\Delta t] \quad i\in[0,N_c-1] \quad (5)$$

where N_k is the number of samples in the window T_k and N_c is the number of time samples of the output signal, i.e., in the window T_c. Equation (5) shows that, using smaller N_k windows, N_c output samples can be correlated as accurate as equation 3.
The number of windows times the number of samples in each window should be equal to the number of samples in the window from 0 to T_α, which we call N_α. We then have:

$$N_wN_k=N_\alpha \quad (6)$$

Equation (5) makes an efficient data processing scheme possible. In this scheme data is buffered for a short period (T_k), as can be seen in equation (5). The buffered data is used to compute the correlation. During the compute time of the correlation new data is buffered.
Note that we have defined four N's, so four lengths:

1. N_α used for the number of samples in total correlation window $(0,T_\alpha)$;
2. N_w used for the total number of smaller windows in the correlation;
3. N_k used for the number of samples in the small correlation window with length T_k;
4. N_c used for the number of samples after the correlation, i.e., of the output signal.

Choice of N_k: We know that the number of output points N_c is limited, so a convenient choice for N_k is: $N_k=2*N_c$. For this choice we can generate N_c output points with the buffered signals. After the computation of the first N_c samples the buffers can be refilled with N_c new samples.
Two straightforward numerical implementations have been made, one which uses copies and one which uses pointer references, to use the data in the buffered arrays. Of course, it is expected that the pointer references will be the faster way, given that the used compiler can generate efficient code with pointer references.

CPU Efficient Correlation

A simple correlation kernel has been used to determine the optimal block size for a certain CPU. We tested the efficiency on three types of hardware, namely on the 64-bit Opteron of AMD, the 64-bit Xeon of Intel, and on the IBM 440d as used in the BlueGene architecture. The main difference lies in how the caches are connected and linked to main memory. On the Intel system, a North Bridge links the two while in the AMD-system this North Bridge is avoided and an on-die memory controller is used. The IBM 440d has a very small L1 and L2 cache and is placed on a 2 core node which share a L3 cache and main memory.
The correlation kernel we used to test the performance is simply:

```
for (l=0; l<ntout; l++) {
    for (j=0; j<nt; j++) {
        C[l] += A[j]*B[j+l];
    }
}
```

We would like to find the value for `nt` which gives the best performance. The code has been tested with values for `nt` varying between 2 and 8388608 points. The `ntout` has also been changed, but does not influence the performance. In Figure 2 the results are show for Xeon, Opteron and 440d. From this figure we can see that the Opteron gives the best performance if the (combined) size of `nt` fits in the L1-cache. The Xeon performs better when the size of `nt` fits in L2-cache. When the data has to come from main memory different results are obtained. The IBM 440d peak performance is below that of the Xeon and Opteron. Note that during these tests the second CPU, which was also present on the boards we used, has not been used. Using this second CPU will give completely different results due to the differences in memory architecture. A detailed explanation about the performance of the CPU's is given in the Appendix (see the second poster).

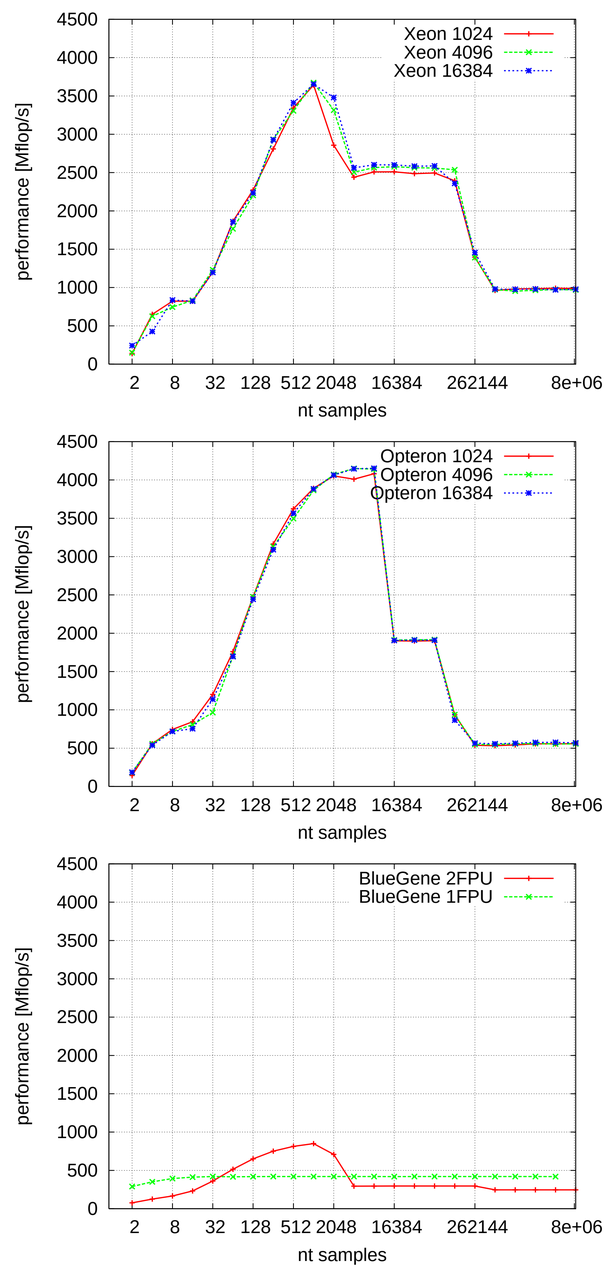


FIGURE 2: *Computational efficiency of a correlation kernel. The performance (in Mflop/s) results for a Xeon 3.0 GHz, an Opteron 2.2 GHz 64K and IBM 440d at 0.7 GHz.*

Some thoughts about other hardware: Besides the standard CPU's we tested there is also a range of other hardware which could be of interest for calculating the correlation: FPGA's, GPU's or special ASIC' like Cell and Clearspead. The main bottleneck in the correlation kernel will be the number of loads (2) of data per the number of floating point operations (2). This means that the connection to main memory or caches must be fast enough to keep up with the floating point calculation rate. For example if an FPGA can do 30 flop at 400 Mhz, it means that the data must be delivered to the FPGA at 48 GB/s!

Conclusion and Discussion

For continuously measured signals the correlation, with a limited number of output samples, can be computed during the recording of new data. Buffers with a limited number of time samples are used to store incoming data during the time the correlation is computed.
To make optimal use of compute hardware the correlation loop should be blocked on the size of the L1/2 cache of the CPU. Different hardware like GPU, FPGA, Cell has not been taken into account, but due to the small flop/load ratio we do not think that this hardware can be efficiently used for the correlation kernel.

Acknowledgment

We would like to acknowledge Advanced Micro Devices (AMD) for giving us an Opteron based server.

References

Draganov, D., Wapenaar, C., and Thorbecke, J. (2006). Seismic interferometry: Reconstructing the earth's reflection response. *Geophysics*, 71(4):SI61–SI70.

Wapenaar, K. and Fokkema, J. (2006). Green's function representations for seismic interferometry. *Geophysics*, 71(4):SI33–SI46.

Efficient computation of passive seismic interferometry

Jan Thorbecke, and Guy Drijkoningen

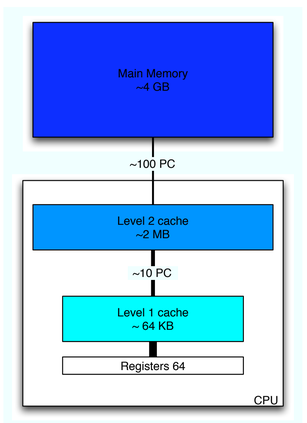
Delft University of Technology, Department of Geotechnology, The Netherlands

Appendix: CPU details

The correlation kernel used to determine the optimal block size for the CPU is

```
for (l=0; l<ntout; l++) {
    for (j=0; j<nt; j++) {
        C[l] += A[j]*B[j+l];
    }
}
```

The code needs to load three 4 Byte floats (assuming load-store architecture) and performs an add and a mult per compute step. This means that the code is limited by the memory bandwidth. For the most inner loop only two loads are needed. The Intel and Pathscale compiler produced code which did not use a load for `C[l]` in this inner loop. The IBM xlc compiler did use a load for the `C` array. The total number of loads for the inner and outer loop are: $nt * ntout + ntout$.
The performance results are shown in Figure 2 (see other poster). To compute the code the Intel compiler (version 9.0) has been used for the Xeon and the Opteron. The IBM xlc C compiler was used to compile for the 440d. The code has been run for three different values of the ntout parameter: 1024, 4096 and 16384. The differences in cache size and memory latencies between the three CPU's are shown in Figure 3.



Feature	Xeon	Opteron	440d
Clock (MHz)	3.0	2.2	0.7
L1-cache	16KB	64KB	32KB
L2-cache	2MB	1MB	2KB
L3-cache	-	-	4MB
latency L1	4	3	8
latency L2	10	12	8
latency L3	-	-	40
latency main	100	150	85
latency main ns	125 (0.8)	68 (2.2)	242 (.35)
peak Gflops	6.0	4.4	2.8

FIGURE 3: A general block-diagram of a modern CPU showing the memory hierarchy. The differences in cache size, latency and peak performance between the three tested CPU's.

Let us first focus on the Xeon chip from Intel. We used the Intel compiler icc with options -O3 -axP. We assume the code can load one 4 Byte float per cycle (1/(3 GHz) s.) from the L1-cache. The maximum peak for this code on the Xeon is then 1/2*peak performance = 3.0 Gflop/s. In Figure 2 the maximum observed peak is 3.6 Gflop/s. The observed performance peak indicates that the compiler produced vectorised code (SSE instructions) and computes 2 floats in 1 cycle. This vectorization is also reported back by the compiler:

```
corrOpt.c(103) : (col. 4) remark: LOOP WAS VECTORIZED.
corrOpt.c(110) : (col. 4) remark: loop was not vectorized:
                                  not inner loop.
corrOpt.c(112) : (col. 5) remark: LOOP WAS VECTORIZED.
```

In Figure 2 we see a first drop in performance above 1024 samples. This is due to the L1-cache whose size is 16K and can contain 3x1365 4 Byte floats. The next drop in performance is from L2 (2 MB) to main memory, occurring above 131072 samples. For the correlation we should aim at block-sizes between the L1 and L2 cache, in this case 1024 samples.
Pseudo code generated by the Intel compiler is shown below. In this pseudo code the use of xmm register and `<..>ps` instructions indicates that the SSE instruction set has been used. (the ps extension of the instruction name indicates parallel (SIMD) use).

```
..B2.73:                        # Preds ..B2.72
        movl      140(%esp), %edi
                                # LOE eax edx ecx ebx esi edi xmm0 xmm1
..B2.74:                        # Preds ..B2.74 ..B2.73
        movaps    (%edi,%esi,4), %xmm2
        movaps    16(%edi,%esi,4), %xmm3
        mulps     (%ebx,%esi,4), %xmm2
        mulps     16(%ebx,%esi,4), %xmm3
        addps     %xmm2, %xmm0
        addps     %xmm3, %xmm1
        addl      $8, %esi
        cmpl      %edx, %esi
        jb        ..B2.74       # Prob 97%
        jmp       ..B2.79       # Prob 100%
        .align    4,0x90
                                # LOE eax edx ecx ebx esi edi xmm0 xmm1
..B2.76:                        # Preds ..B2.72
        movl      140(%esp), %edi
        .align    4,0x90
```

Let us next focus on the Opteron 248 (2.2GHz) from AMD, as shown in the middle graph in Figure 2. The Opteron has a similar L1/L2 cache structure as the Xeon, but the connection to main memory is different. Where the Xeon has to go to the NorthBridge to go to main memory, the Opteron uses an on-die memory controller. Two HyperTransport ports to the DDR 400 memory dimms gives a bandwidth of 6.4 GB/s. The L1 cache of the Opteron is 64K and the L2 cache is 1 MB.
For the compilation Intel's IA-32 compiler (/opt/intel/cc/9.0/bin/icc -O3 -axN -vec-report2 -c corrOpt.c) was used (some dummy __intel_functions were defined and the produced .o file was linked using the Pathscale compiler). As can be seen on the figure, the performance reached (above 4.0 Gflop/s) is not only higher than the Xeon, but it also reached at larger block size, namely at 8192 samples. The next drop drom the L2 cache to the main memory occurs above 65536, so earlier than for the Xeon. So from this graph it can be seen that the correlation runs optimally at 8192 samples.

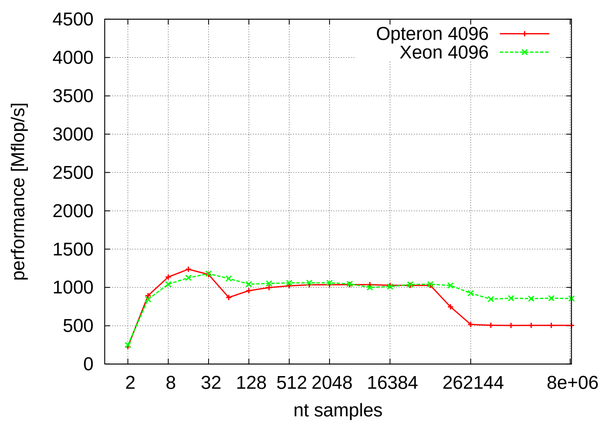


FIGURE 4: Computational efficiency of correlation using pathscale compiler

For the generation of the results in Figure 4 the Pathscale compiler has been used for both the Xeon and the Opteron. The compiler is pathcc (2.4) with options `-O3 -OPT:Ofast -OPT:alias=disjoint` This compiler does not vectorize the most inner loops, while the Intel compiler does vectorize these loops. The Pathscale compile does show nicely (see below) how it optimise the inner loop. It is noteworthy that the performance is now well below the performance with the compiler from Intel; it reaches a peak performance below 1.5 Gflop/s. Interestingly the peak performance of the Opteron is around 1 Gflop/s when the data has to come from main memory. This is the same performance as the code the Intel compiler produced for the Xeon. Note that using compiler directives in the code the Pathscale compiler should also be able to vectorise the most inner loops.

```
#<loop> Loop body line 108, nesting depth: 5
#<sched>
#<sched> Loop schedule length: 31 cycles
                    (ignoring nested loops)
#<sched>
#<sched>    15 flops        ( 24% of peak)
#<sched>     7 mem refs     ( 11% of peak)
#<sched>     4 integer ops  (  6% of peak)
#<sched>    20 instructions ( 16% of peak)
#<sched>
#<freq>
#<freq> BB:58 frequency = 52857164.00000 (heuristic)
#<freq> BB:58 => BB:59 probability = 0.01000
#<freq> BB:58 => BB:58 probability = 0.99000
#<freq>
        movaps %xmm9,%xmm5              # [0]
        mulss 0(%r8),%xmm5              # [2]
        addss 0(%rcx),%xmm5             # [8]
        movaps %xmm7,%xmm4              # [6]
        mulss 4(%r8),%xmm4              # [8]
        movaps %xmm8,%xmm3              # [10]
        mulss 8(%r8),%xmm3              # [12]
        addss %xmm5,%xmm4               # [14]
        movaps %xmm6,%xmm2              # [13]
        mulss 12(%r8),%xmm2             # [16]
        addss %xmm4,%xmm3               # [18]
        movaps %xmm1,%xmm0              # [17]
        mulss 16(%r8),%xmm0             # [20]
        addss %xmm3,%xmm2               # [22]
        addss %xmm2,%xmm0               # [26]
        addq $4,%rcx                    # [28]
        addq $4,%r8                     # [29]
        cmpq %r9,%rcx                   # [29]
        movss %xmm0,-4(%rcx)            # [30]
```

The Blue Gene 440d CPU results are shown in the bottom graph of Figure 2. A zoom of the results is shown in Figure 4. The 440d has an second FPU unit which can be used for the computations. This second FPU can only be used if the computations are done in double precision and the arrays aligned on 16-bit boundaries. A quadword (i.e., 128 bits) datapath between the PPC 440s Data Cache and the PPC 440 FP2 allows for dual data elements (either double-precision or single precision) to be loaded or stored each cycle.

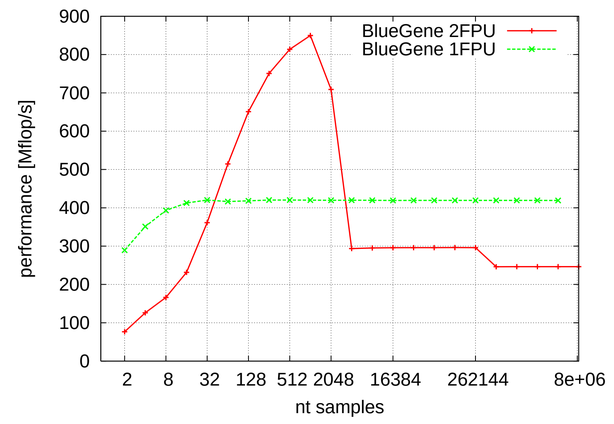


FIGURE 5: Zoom in of the Blue Gene results

As mentioned before the IBM xlc compiler also load the `C[l]` element in the inner loop, to avoid this load a temporary variable has been introduced (see below for the changed loop). With this change, and making the arrays double precision, the code could use the 2'nd FPU. The peak obtained with the 2'nd FPU is at 0.8 Gflop/s. In the compiled block shown below it is observed that in the inner loop the compiler generated code with 8 LFPL(U) (load) and 4 FPMADD instructions. The latency of the FPMADD instruction in 5 cycles and has a repeat rate of 1 cycle. This explains the 0.8 Gflop/s. The quadword load instructions have been used (instruction `lfpdx`), but cache misses of the `B[j+l]` will slow down the performance. If the loop changes to `cr += A[j]*B[j]` the performance increased to 1.2 Gflop/s. Interesting is that in using only one FPU the code performs better (0.4 Gflop/s) than using 2 FPU's (0.3 Gflop/s), when it has to get data outside the cache . The virtual node mode has not been tested.

```
for (l=0; l<ntout; l++) {
     cr = 0.0;
     for (j=0; j<nt; j++) {
          cr += A[j]*B[j+l];
     }
     C[l] = cr;
}

mpixlc -O5 -qarch=440d -qhot -qfloat=norngchk \
       -qlist -qlistopt -qreport   -c corrOpt.c

(I) <SIMD info> SIMDIZABLE (Loop index 4 on line 131
       with nest-level 3 and iteration count 100.)

Examine loop <4> on line 132 in file "corrOpt.c"(simdizable)
[reduct][versioned(relative-align natural-align tripcount)]

                                CL.745:
 0006E4 lfpdx    7C34239C   1     LFPL
        fp1,fp33=$.V.A[]0.rns42.1(gr20,gr4,0,trap=16)
 0006E8 fpmadd   010A4760   1     FPMADD
        fp8,fp40=fp8,fp40,fp10,fp42,fp29,fp61,fcr
 0006EC lfpdx    7D96239C   1     LFPL
        fp12,fp44=$.V.B[]0.rns41.0(gr22,gr4,0,trap=16)
 0006F0 lfpdx    7C142B9C   1     LFPL
        fp0,fp32=$.V.A[]0.rns42.1(gr20,gr5,0,trap=32)
 0006F4 fpmadd   00E53B60   1     FPMADD
        fp7,fp39=fp7,fp39,fp5,fp37,fp13,fp45,fcr
 0006F8 lfpdx    7D762B9C   1     LFPL
        fp11,fp43=$.V.B[]0.rns41.0(gr22,gr5,0,trap=32)
 0006FC lfpdx    7D54339C   1     LFPL
        fp10,fp42=$.V.A[]0.rns42.1(gr20,gr6,0,trap=48)
 000700 fpmadd   01214B20   1     FPMADD
        fp9,fp41=fp9,fp41,fp1,fp33,fp12,fp44,fcr
 000704 lfpdx    7FB6339C   1     LFPL
        fp29,fp61=$.V.B[]0.rns41.0(gr22,gr6,0,trap=48)
 000708 lfpdux   7CB43BDC   1     LFPLU
        fp5,fp37,gr20=$.V.A[]0.rns42.1(gr20,gr7,0,trap=64)
 00070C fpmadd   00C032E0   1     FPMADD
        fp6,fp38=fp6,fp38,fp0,fp32,fp11,fp43,fcr
 000710 lfpdux   7DB63BDC   1     LFPLU
        fp13,fp45,gr22=$.V.B[]0.rns41.0(gr22,gr7,0,trap=64)
 000714 bc       4320FFD0   0     BCT
          ctr=CL.745,taken=100%(100,0)
                               CL.744:
```

In the performed tests only one CPU was used and competing for main memory by other processors is not taken into account. Also parallelization was not considered. However, the algorithm can easily be made parallel without large communication costs between the processors. During parallel correlation the data throughput through the machine will be significant and the interconnect architecture of the hardware will become much more important. We expect that the Blue Gene architecture will be an advantage compared to standard Linux clusters.