

Report 3 - Acceleration

Fluerătoru Laura

1 Introduction

This report outlines the steps undertaken to accelerate several parts of the MUSIC algorithm. Through profiling, there were found two hotspots in the chosen implementation, in computing the MUSIC spectrum and in the block that performs the autocorrelation of the signal. The former aspect is dealt with in Section §2, while the latter is to be discussed in Section §3. Some conclusions and future work are found in Section §4 and the code can be found in Section §5.

2 Accelerating the computation of the MUSIC spectrum

The MUSIC spectrum is computed in the **MUSIC Linear Array** block, whose functionality was detailed in Report 2, Section 4.4. Through profiling, it was found that a computationally intensive part is in the vectorial multiplication used in the denominator of the MUSIC spectrum in formula (1).

$$P_{MUSIC}(\theta) = \frac{1}{\mathbf{a}^H(\theta) \mathbf{V}_N \mathbf{V}_N^H \mathbf{a}(\theta)} \quad (1)$$

In the above equation, $\mathbf{a}(\theta)$ is the basis vector of the signal subspace, of length M, where M is the size of the antenna array, while \mathbf{V}_N is a matrix formed by the eigenvectors of the autocorrelation matrix \mathbf{R}_{xx} that are orthogonal to the steering vectors. The product $\mathbf{V}_N \mathbf{V}_N^H$ is therefore a square matrix of size M by M, which we will further denote as \mathbf{V}_{Nsq} . Its range space is what we call the noise subspace [1].

Both $\mathbf{a}(\theta)$ and \mathbf{V}_{Nsq} are matrices of complex elements. The real and imaginary parts of $\mathbf{a}(\theta)$ are each in the $[-1, 1]$ interval. As of \mathbf{V}_{Nsq} , its range of values is difficult to establish, but in practice it has been observed that it belongs to the same interval. Therefore, for the first part of the multiplication, $\mathbf{a}^H(\theta) \mathbf{V}_{Nsq}$, the real and imaginary parts of an element of the result cannot exceed the range $[-M, M]$. This result is a row vector of size 1 by M, which we denote $X_{1,N}$.

We have decided to compute $X_{1,N}$ on the ConnexArray and the final part, $X_{1,N} \mathbf{a}(\theta)$, on the ARM processor, due to performance considerations that will be outlined.

2.1 Multiplication kernel on the ConnexArray SIMD

First, we consider the multiplication of row vector of length M by a square matrix of size M by M, which yields as a result a row vector of length M.

$$X_{1,N} \triangleq \begin{bmatrix} a_0 & a_1 & \dots & a_{M-1} \end{bmatrix} \begin{bmatrix} v_{0,0} & v_{0,1} & \dots & v_{0,M-1} \\ v_{1,0} & v_{1,1} & \dots & v_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{M-1,0} & v_{M-1,1} & \dots & v_{M-1,M-1} \end{bmatrix} \quad (2)$$

$$X_{1,N} = \left[\sum_{i=0}^{M-1} a_i v_{i,0} \quad \sum_{i=0}^{M-1} a_i v_{i,1} \quad \dots \quad \sum_{i=0}^{M-1} a_i v_{i,M-1} \right], \text{ where} \quad (3)$$

$$a_i = x_i + jy_i, \quad i = \overline{0, M-1} \quad (4)$$

$$v_{i,k} = x_{i,k} + jy_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (5)$$

The result can be further written as following:

$$X_{1,N} = \left[\begin{array}{c} \sum_{i=0}^{M-1} (x_i x_{i,0} - y_i y_{i,0}) + j \sum_{i=0}^{M-1} (x_i y_{i,0} + y_i x_{i,0}) \\ \sum_{i=0}^{M-1} (x_i x_{i,1} - y_i y_{i,1}) + j \sum_{i=0}^{M-1} (x_i y_{i,1} + y_i x_{i,1}) \\ \dots \\ \sum_{i=0}^{M-1} (x_i x_{i,M-1} - y_i y_{i,M-1}) + j \sum_{i=0}^{M-1} (x_i y_{i,M-1} + y_i x_{i,M-1}) \end{array} \right]^T \quad (6)$$

The proposed way for computing the result makes use of the arrangement in Figure 1 in the ConnexArray processor.

	PE ₀	PE ₁	PE ₂	PE ₃	...	PE _{2M-2}
R1	x ₀	y ₀	x ₁	y ₁	...	x _{M-1}
R2	x _{0,0}	y _{0,0}	x _{1,0}	y _{1,0}	...	x _{M-1,0}
R3	x ₀ x _{0,0}	x ₁ y _{0,0}	x ₁ x _{1,0}	y ₁ y _{1,0}	...	x _{M-1} x _{M-1,0}
R4	x ₀ y _{0,0}		x ₁ y _{1,0}		...	x _{M-1} y _{M-1,0}
R5		y ₀ x _{0,0}		y ₁ x _{1,0}	...	

Figure 1: Arranging the elements in the ConnexArray processing elements

First, the input array and the input matrix are loaded into the R1 and R2 registers, respectively. The real part of an element will be followed by the imaginary part of the same element, so for each complex element two PEs are needed. In the case of the input array, since its elements will have to be multiplied by each column of the input matrix, they are loaded in blocks of size M, successively, as shown above. Then, in each PE, we compute the product of the registers R1 and R2 and store it in R3, thus obtaining the intermediary products necessary for the real part of the result's elements.

To obtain the intermediary products for the imaginary parts, we first shift R2 to the left with one position and store the result in R4. By multiplying R1 with R4, in half of the PEs (the ones with an even index) we obtain the products in (8) and store them in R4.

$$x_i y_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (7)$$

Secondly, we shift R2 to the right with one position and store the result in R5. By multiplying R1 by R5, we obtain the intermediary products in (8) in the PEs with an odd index and store them in R5.

$$y_i x_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (8)$$

In the figure, the data of interest in the registers is marked by a green background, and the unnecessary data by a red one.

Next, we move the data in R5 in the PEs with odd index in register R4, so we now have all the intermediary products for the real and imaginary parts in R3 and R4, respectively. By performing successive reductions on blocks of size $2M$ on the aforementioned registers, we obtain the real and imaginary parts of the elements of $X_{1,N}$.

Listing 1 contains the kernel that implements the described algorithm. The kernel is contained in a function that takes as parameters:

- `process_at_once`, a parameter used in case we successively multiply more chunks of data
- `size_of_block`, the size of a block on which reduction is performed at once
- `blocks_to_reduce`, on how many blocks the reduction is performed

```

1 void multiply_kernel(int process_at_once, int size_of_block, int
  blocks_to_reduce)
2 {
3   BEGIN_KERNEL("multiply_arr_mat");
4   EXECUTE_IN_ALL(
5     R25 = 0;
6     R26 = 511;
7     R30 = 1;
8     R31 = 0;
9     R28 = size_of_block;
10  )
11
12  EXECUTE_IN_ALL(
13    R1 = LS[R25];           // z1 = a1 + j * b1
14    R2 = LS[R26];           // z2 = a2 + j * b2
15    R29 = INDEX;           // Used later to select PEs for reduction
16    R27 = size_of_block;    // Used to select blocks of ARR_SIZE_C for reduction
17
18    R3 = R1 * R2;           // a1 * a2, b1 * b2
19    R3 = MULT_HIGH();
20
21    CELL_SHL(R2, R30);      // Bring b2 to the left to calc b2 * a1
22    NOP;
23    R4 = SHIFT_REG;
24    R4 = R1 * R4;           // a1 * b2
25    R4 = MULT_HIGH();
26
27    CELL_SHR(R2, R30);
28    NOP;
29    R5 = SHIFT_REG;
30    R5 = R1 * R5;           // b1 * a2
31    R5 = MULT_HIGH();
32
33    R9 = INDEX;             // Select only the odd PEs
34    R9 = R9 & R30;
35    R7 = (R9 == R30);
36  )
37
38  EXECUTE_WHERE_EQ(         // Only in the odd PEs
39    // Getting -b1 * b2 in each odd cell
40    R3 = R31 - R3;          // All partial real parts are in R3
41
42    R4 = R5;                // All partial imaginary parts are now in R4
43  )

```

```

44
45 REPEAT_X_TIMES(blocks_to_reduce);
46 EXECUTE_IN_ALL(
47     R7 = (R29 < R27);    // Select only blocks of 8 PEs at a time by
48                           // checking that the index is < k * 8
49 )
50 EXECUTE_WHERE_LT(
51     R29 = 129;           // A random number > 128 so these PEs won't be
52                           // selected again
53     REDUCE(R3);          // Real part
54     REDUCE(R4);          // Imaginary part
55 )
56 EXECUTE_IN_ALL(
57     R27 = R27 + R28;     // Go to the next block of 8 PEs
58 )
59 END_REPEAT;
60 END_KERNEL("multiply_arr_mat");
61 }

```

Listing 1: Kernel for multiplying a row vector with a matrix

2.2 Integrating the kernel in a standalone GNURadio out-of-tree module

In the MUSIC algorithm, the MUSIC spectrum will have a minimum in a point corresponding to the angle of arrival. An array manifold vector is generated for a number of `d_spectrum_length` angles spaced evenly between 0 and 180 degrees, which will be used in (1). Therefore, for each input signal we will compute `d_spectrum_length` values of the MUSIC spectrum. This also means that the matrix $\mathbf{V}_{N_{sq}}$ will be a factor in just as many products.

In order to assert the functionality of the kernel, we have created a standalone module named `multiply_cc` that takes as inputs a number of `NR_ARRAYS` steering vectors of size 1 by `ARR_SIZE` and a square matrix of size `MAT_SIZE` and outputs the results of their product, of size `NR_ARRAYS` by `ARR_SIZE`. In reality, we cannot have OOT (out-of-tree) modules with a variable number of inputs and outputs, so we linearize the inputs and the outputs. The input matrix has a column-by-column order, since this is the way matrices are stored in the libraries Armadillo [2] and BLAS[3]. The inputs arrays are row arrays, so they are considered to be read array-by-array, and the same is also true for the output.

The module uses a `forecast` method to ensure that, for an output item, enough input items are available, according to the correspondence above. The module is part of a graph that consists of two vector sources, a throttle block, the `multiply_cc` block and a vector sink. The program that runs the graph feeds a set of inputs with random values generated in the GNU Octave [4] environment and asserts the results by comparing them with the expected ones computed in Octave.

Since the Connex Array has a capacity of `VECTOR_ARRAY_SIZE = 128` PEs, we will not be able to accomodate inputs with an arbitrarily large number of elements. The way we proceed is to separate, for each output element that needs to be produced, the input elements into chunks. How many input arrays can be multiplied by the same matrix in one job on the ConnexArray depends on the size of the matrix `MAT_SIZE` and is equal to `VECTOR_ARRAY_SIZE / MAT_SIZE`. A job is considered to be an execution of the kernel on the ConnexArray.

Before each job, the input data has to be *prepared*, meaning that it has to be scaled and saved into an array of type `uint16_t`. The input matrix is prepared only once for each output item in the member function `prepareInMatConnex`, while the input arrays have to be prepared for each chunk using the member function `prepareInArrConnex`.

After we have the input data in arrays of integers ready, we can launch the kernel execution. We obtain the output data by reading a number of elements from the reduction queue. The number of elements read is equal to two times the number of output items computed in a chunk (because we read a real and an imaginary part).

Once the data is ready, it has to be scaled back and converted to the type `gr_complex`, which is an alias for the type `std::complex<float>`, used for the output elements.

An important aspect to note is that in this scenario, once the kernel execution is launched, the program continues executing the instruction in its main thread. The method that reads the reduction results has to ensure that the ConnexArray has finished producing the desired number of results and therefore the main thread will be in a blocking state until that processing is finished. The time spent waiting for the ConnexArray to finish its execution could be spent doing other processing in the main thread by using multiple threads of execution.

2.3 Multithreading support for the kernel

We can take advantage of the time spent with the execution of the kernel by doing other necessary processing in the main thread. This processing can be either preparing the elements for the next chunk, preparing the output for the past chunk or both. Also, with the aid of the kernel we obtain only the intermediary matrix $\mathbf{X}_{1 \times N}$, but the final results needs an additional multiplication of this result by the steering vector $\mathbf{a}(\theta)$, and this could also be computed while waiting for the results on the SIMD processor.

A pseudocode that better explains the flow of the program is the following:

```

1
2 prepare_current_data();
3 for all chunks:
4     launch_kernel_execution();
5     if not last chunk:
6         prepare_next_data();
7     if not first chunk:
8         process_past_data();
9     join_threads();
10    read_results();
11    increment_pointers();
12 process_past_data();

```

Listing 2: Pseudocode explaining the flow of the multithreading program

The full code for the module that implements the multiplication with multithreading support can be found in Listing 3 in Section §5.

Informal comment: the way I did this was by launching a new thread with the kernel function, then preparing the elements for the next chunk, processing the elements of the past chunk and then joining the threads before reading the reduction, but now I see that it is kind of pointless, since the reduction waits anyway for the execution of the kernel to complete and launching the

execution in itself is not blocking. Was there another way of using multiple threads that could give better results?

3 Accelerating the computation of the autocorrelation

TODO

4 Conclusion

Further things to do:

1. Rethink the multithreading part (need some help here)
2. Explain how and why I eliminated the first hotspot (that I considered redundant). Should I email the guys that made the graph and ask them about it? I seriously doubt they remember anymore.
3. Include profiling data after eliminating the hotspot above and draw some conclusions
4. Profile on ARM (and hope the results coincide with the ones above)
5. Test the kernel on ARM + Connex
6. Integrate the kernel in the graph, test it, profile it and compare
7. For testing in 5: Create some files with random data generated in Octave along with the expected results and maybe integrate with gtest?
8. Tackle the autocorrelation part. I was thinking that even if the hotspot in the autocorrelation part seems to be the memcpy, maybe I can do the whole autocorrelation on Connex, too. After all it's also a matrix/array multiplication and maybe it isn't that hard to reuse what I already have. And this way I can say that I also have something generic to do on Connex (autocorrelation is pretty important in a lot of algorithms) that is not necessarily related to MUSIC and can be reused in other cases, too.
9. Decide what parts from the past reports will be in the final thesis and assemble them. I think I'll have enough pages, so that shouldn't be the problem, but instead choose what's important to include.
10. Less important but still: ask someone about the style of the code in the thesis. The official template that I have draws ugly boxes around references and I'm not sure it's ok. Also, I don't know about the style of the code included.

5 Code snippets

5.1 Accelerating the matrix multiplication - multithreading

```
1 #ifndef HAVE_CONFIG_H
2 #include "config.h"
3 #endif
4
5 #include <gnuradio/io_signature.h>
6 #include "multiply_cc_impl.h"
7 #include <cmath>
8 #include <thread>
9
10 namespace gr {
11     namespace opincaa {
12
13         /*
14          * Kernel related functions
15          */
16         void executeMultiplyArrMat(ConnexMachine *connex);
17         void multiply_kernel(
18             int process_at_once,
19             int size_of_block,
20             int blocks_to_reduce);
21         void multiply_once(int size_of_block, int blocks_to_reduce);
22
23         multiply_cc::sptr
24         multiply_cc::make(
25             std::string distributionFIFO,
26             std::string reductionFIFO,
27             std::string writeFIFO,
28             std::string readFIFO)
29         {
30             return gnuradio::get_initial_sptr
31                 (new multiply_cc_impl(distributionFIFO,
32                                     reductionFIFO,
33                                     writeFIFO,
34                                     readFIFO));
35         }
36
37         /*
38          * The private constructor
39          */
40         multiply_cc_impl::multiply_cc_impl(
41             std::string distributionFIFO,
42             std::string reductionFIFO,
43             std::string writeFIFO,
44             std::string readFIFO)
45             : gr::sync_block("multiply_cc",
46                             gr::io_signature::make(2, 2, sizeof(gr_complex) * MAT_SIZE),
47                             gr::io_signature::make(1, 1, sizeof(gr_complex) * NR_ARRAYS_ELEMS)
48                             )
49         {
50             try {
51                 connex = new ConnexMachine(distributionFIFO,
52                                             reductionFIFO,
53                                             writeFIFO,
54                                             readFIFO);
55             } catch (std::string err) {
56                 std::cout << err << std::endl;
57             }
58         }
59     }
60 }
```

```

56     }
57
58     const int blocks_to_reduce = VECTOR_ARRAY_SIZE / ARR_SIZE_C;
59     const int size_of_block = ARR_SIZE_C;
60
61     factor_mult1 = 1 << 14;
62     factor_mult2 = 1 << 16;
63     factor_res = 1 << 14;
64
65     multiply_kernel(PROCESS_AT_ONCE, size_of_block, blocks_to_reduce);
66 }
67
68 /*
69  * Our virtual destructor.
70  */
71 multiply_cc_impl::~multiply_cc_impl()
72 {
73     delete connex;
74 }
75
76 void multiply_cc_impl::forecast(
77     int noutput_items,
78     gr_vector_int &ninput_items_required)
79 {
80     // The first input is an array of ARR_SIZE elements, while the second is
81     // a linearized, column by column matrix of ARR_SIZE x ARR_SIZE elements
82     // The result is a 4 array element representing the multiplication between
83     // the input array and input matrix.
84     ninput_items_required[0] = (NR_ARRAYS / MAT_SIZE) * noutput_items;
85     ninput_items_required[1] = noutput_items;
86 }
87
88 int
89 multiply_cc_impl::work(int noutput_items,
90     gr_vector_const_void_star &input_items,
91     gr_vector_void_star &output_items)
92 {
93     if (!connex) {
94         return noutput_items;
95     }
96
97     const gr_complex *in0 = reinterpret_cast<const gr_complex *>(input_items
98         [0]);
99     const gr_complex *in1 = reinterpret_cast<const gr_complex *>(input_items
100         [1]);
101     gr_complex *out = reinterpret_cast<gr_complex *>(output_items[0]);
102
103     std::cout << "Called worker with " << noutput_items << " out items." <<
104         std::endl;
105
106     for (int i = 0; i < noutput_items; i++) {
107         std::cout << "Output item nr. " << i << std::endl;
108         const gr_complex *in0_round = &in0[i * NR_ARRAYS_ELEMS];
109         const gr_complex *in1_round = &in1[i * MAT_SIZE];
110         gr_complex *out_round = &out[i * NR_ARRAYS_ELEMS];
111
112         std::vector<gr_complex> final_res(NR_ARRAYS, 0);
113
114         const int nr_chunks = NR_ARRAYS / ARR_IN_CHUNK;
115         const int nr_elem_chunk = ARR_IN_CHUNK * ARR_SIZE;

```



```

113     const int nr_elems_calc = PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE /
        ARR_SIZE_C;
114
115     uint16_t *in0_i = (uint16_t *)
        malloc(nr_chunks * VECTOR_ARRAY_SIZE * sizeof(uint16_t));
116     uint16_t *in1_i = (uint16_t *)
        malloc(VECTOR_ARRAY_SIZE * sizeof(uint16_t));
117     int32_t *res_mult = (int32_t *)
        malloc(nr_chunks * VECTOR_ARRAY_SIZE * sizeof(int32_t));
118
119     if ((in0_i == NULL) || (in1_i == NULL) || (res_mult == NULL))
120         std::cout << "Malloc error!" << std::endl;
121
122     uint16_t *in0_curr, *in1_curr, *in0_next;
123
124     // Prepare the first chunk
125     in0_curr = in0_i;
126     in1_curr = in1_i;
127
128     int32_t *res_curr_chunk, *res_past_chunk = NULL;
129     const gr_complex *arr_next_chunk, *arr_past_chunk;
130
131     const gr_complex *arr_curr_chunk = in0_round;
132     const gr_complex *mat = in1_round; // Using the same mat for all chunks
133     gr_complex *out_curr_chunk = out_round, *out_past_chunk;
134
135     const int elems_to_prepare = PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE / 2;
136
137     prepareInArrConnex(in0_curr, arr_curr_chunk, elems_to_prepare);
138     prepareInMatConnex(in1_curr, mat, elems_to_prepare);
139
140     for (int cnt_chunk = 0; cnt_chunk < nr_chunks; cnt_chunk++) {
141         res_curr_chunk = &res_mult[cnt_chunk * VECTOR_ARRAY_SIZE];
142
143         connex->writeDataToArray(in0_curr, PROCESS_AT_ONCE, 0);
144         connex->writeDataToArray(in1_curr, PROCESS_AT_ONCE, 511);
145
146         std::thread t(executeMultiplyArrMat, connex);
147
148         try {
149             // Prepare future data for all but the last chunk
150             if (cnt_chunk != nr_chunks - 1) {
151                 in0_next = in0_curr + VECTOR_ARRAY_SIZE;
152                 arr_next_chunk = arr_curr_chunk + nr_elem_chunk;
153
154                 prepareInArrConnex(in0_next, arr_next_chunk, elems_to_prepare);
155             }
156
157             // Process past data for all but the first chunk
158             if (cnt_chunk != 0) {
159                 out_past_chunk = &out_round[(cnt_chunk - 1) * nr_elem_chunk];
160                 prepareOutDataConnex(out_past_chunk, res_past_chunk, nr_elems_calc);
161                 multLineCol(final_res, out_past_chunk, arr_past_chunk, cnt_chunk - 1);
162             }
163         } catch (...) {
164             t.join();
165             throw;
166         }
167     }
168 }

```

```

170
171 // Must join threads before reading the reduction
172 t.join();
173
174 // 2 * —> complex elements, so we have real *and* imag parts
175 connex->readMultiReduction(2 * nr_elems_calc, res_curr_chunk);
176
177 // Incrementing the pointers for the next chunk
178 in0_curr = in0_next;
179 arr_past_chunk = arr_curr_chunk;
180 arr_curr_chunk = arr_next_chunk;
181 res_past_chunk = res_curr_chunk;
182 } // end loop for each chunk
183
184 // Results for the last chunk
185 out_past_chunk = &out_round[(nr_chunks - 1) * nr_elem_chunk];
186 prepareOutDataConnex(out_past_chunk, res_past_chunk, nr_elems_calc);
187 multLineCol(final_res, out_past_chunk, arr_past_chunk, nr_chunks - 1);
188
189 free(in0_i);
190 free(in1_i);
191 free(res_mult);
192 } // end loop for each output item
193
194 return noutput_items;
195 }
196
197 void multiply_cc_impl::prepareInArrConnex(
198     uint16_t *out_arr, const gr_complex *in_arr, const int nr_elems)
199 {
200     for (int j = 0; j < nr_elems; j++) {
201         out_arr[2 * j] = static_cast<uint16_t>
202             (in_arr[(j / 16) * ARR_SIZE + (j % 4)].real() * factor_mult1);
203         out_arr[2 * j + 1] = static_cast<uint16_t>
204             (in_arr[(j / 16) * ARR_SIZE + (j % 4)].imag() * factor_mult1);
205     }
206 }
207
208 void multiply_cc_impl::prepareInMatConnex(
209     uint16_t *out_mat, const gr_complex *in_mat, const int nr_elems)
210 {
211     for (int j = 0; j < nr_elems; j++) {
212         out_mat[2 * j] = static_cast<uint16_t>
213             (in_mat[j % 16].real() * factor_mult2);
214         out_mat[2 * j + 1] = static_cast<uint16_t>
215             (in_mat[j % 16].imag() * factor_mult2);
216     }
217 }
218
219 void multiply_cc_impl::prepareOutDataConnex(
220     gr_complex *out_data,
221     const int32_t *raw_out_data,
222     const int nr_elems)
223 {
224     float temp0, temp1;
225     for (int j = 0; j < nr_elems; j++) {
226         temp0 = (static_cast<float>(raw_out_data[j * 2])) / factor_res;
227         temp1 = (static_cast<float>(raw_out_data[j * 2 + 1])) / factor_res;
228
229         out_data[j] = std::complex<float>(temp0, temp1);

```

```

230 }
231 }
232
233 void multiply_cc_impl::multLineCol(
234     std::vector<gr_complex> result,
235     const gr_complex *line_arr,
236     const gr_complex *col_arr,
237     const int nr_chunk)
238 {
239     // Do the final line array * column array multiplication
240     // As a result, we have an element for each array processed in a chunk
241     std::complex<float> acc(0, 0);
242     for (int j = 0; j < ARR_IN_CHUNK; j++) {
243         acc = (0, 0);
244         for (int jj = 0; jj < ARR_SIZE; jj++) {
245             acc = acc +
246                 (line_arr[j * ARR_IN_CHUNK + jj] * col_arr[j * ARR_IN_CHUNK + jj]);
247         }
248         result[nr_chunk * ARR_IN_CHUNK + j] = acc;
249         std::cout << "result[" << nr_chunk * ARR_IN_CHUNK + j << "] = "
250             << result[nr_chunk * ARR_IN_CHUNK + j] << std::endl;
251     }
252 }
253
254 /*=====
255  * Define ConnexArray kernels that will be used in the worker
256  *=====*/
257 void executeMultiplyArrMat(ConnexMachine *connex)
258 {
259     connex->executeKernel("multiply_arr_mat");
260 }
261
262 void multiply_kernel(int process_at_once, int size_of_block, int
263     blocks_to_reduce)
264 {
265     BEGIN_KERNEL("multiply_arr_mat");
266     EXECUTE_IN_ALL(
267         R25 = 0;
268         R26 = 511;
269         R30 = 1;
270         R31 = 0;
271         R28 = size_of_block; // Equal to ARR_SIZE_C; dimension of the blocks
272                             // on which reduction is performed at once
273     )
274
275     EXECUTE_IN_ALL(
276         R1 = LS[R25]; // z1 = a1 + j * b1
277         R2 = LS[R26]; // z2 = a2 + j * b2
278         R29 = INDEX; // Used later to select PEs for reduction
279         R27 = size_of_block; // Used to select blocks of ARR_SIZE_C for
280                             reduction
281
282         R3 = R1 * R2; // a1 * a2, b1 * b2
283         R3 = MULT_HIGH();
284
285         CELL_SHL(R2, R30); // Bring b2 to the left to calc b2 * a1
286         NOP;
287         R4 = SHIFT_REG;
288         R4 = R1 * R4; // a1 * b2
289         R4 = MULT_HIGH();

```

```

288
289     CELL_SHR(R2, R30);
290     NOP;
291     R5 = SHIFT_REG;
292     R5 = R1 * R5;           // b1 * a2
293     R5 = MULT_HIGH();
294
295     R9 = INDEX;           // Select only the odd PEs
296     R9 = R9 & R30;
297     R7 = (R9 == R30);
298 )
299
300 EXECUTE_WHERE_EQ(         // Only in the odd PEs
301     // Getting -b1 * b2 in each odd cell
302     R3 = R31 - R3;        // All partial real parts are in R3
303
304     R4 = R5;              // All partial imaginary parts are now in R4
305 )
306
307 REPEAT_X_TIMES(blocks_to_reduce);
308 EXECUTE_IN_ALL(
309     R7 = (R29 < R27);    // Select only blocks of 8 PEs at a time by
310                          // checking that the index is < k * 8
311 )
312 EXECUTE_WHERE_LT(
313     R29 = 129;           // A random number > 128 so these PEs won't be
314                          // selected again
315     REDUCE(R3);           // Real part
316     REDUCE(R4);           // Imaginary part
317 )
318 EXECUTE_IN_ALL(
319     R27 = R27 + R28;     // Go to the next block of 8 PEs
320 )
321 END_REPEAT;
322
323 END_KERNEL("multiply_arr_mat");
324 }
325 } /* namespace opincaa */
326 } /* namespace gr */

```

Listing 3: The C++ code used for accelerating the matrix multiplication in the MUSIC Linear Array block - using a separate thread for launching the ConnexArray job

```

1 #ifndef INCLUDED_OPINCAA_MULTIPLY_CC_IMPL_H
2 #define INCLUDED_OPINCAA_MULTIPLY_CC_IMPL_H
3
4 #include <opincaa/multiply_cc.h>
5 #include "ConnexMachine.h"
6
7 #define VECTOR_ARRAY_SIZE 128
8 #define ARR_SIZE 4
9 #define MAT_SIZE (ARR_SIZE * ARR_SIZE)
10
11 #define ARR_SIZE_C (ARR_SIZE * 2)
12 #define MAT_SIZE_C (MAT_SIZE * 2)
13
14 #define PROCESS_AT_ONCE 1
15
16 // Defining a cluster of 16 arrays of 4 elements each that are multiplied with
17 // the same matrix
18 #define NR_ARRAYS 32

```

```

19 #define NR_ARRAYS_ELEMS (NR_ARRAYS * ARR_SIZE)
20
21 // how many arr * mat multiplications can be performed on the ConnexArray in a
22 // round aka. in a chunk
23 #define ARR_IN_CHUNK (PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE / MAT_SIZE_C)
24
25 #define MIN(x, y) ((x > y) ? y : x)
26
27
28 namespace gr {
29     namespace opincaa {
30
31         class multiply_cc_impl : public multiply_cc
32         {
33         private:
34             ConnexMachine *connex;
35
36             // Factors required for scaling the data
37             int factor_mult1;
38             int factor_mult2;
39             int factor_res;
40
41             void forecast(
42                 int noutput_items,
43                 gr_vector_int &ninput_items_required);
44
45             /* \brief scales and casts a number of nr_elems input elements in an array
46              *      of size ARR_SIZE
47              */
48             void prepareInArrConnex(
49                 uint16_t *out_arr, const gr_complex *in_arr, const int nr_elems);
50
51             /* \brief scales and casts a number of nr_elems input elements in an
52              *      matrix of size MAT_SIZE x MAT_SIZE
53              */
54             void prepareInMatConnex(
55                 uint16_t *out_mat, const gr_complex *in_mat, const int nr_elems);
56
57             /* \brief Scales back the output data and converts the array of
58              *      real and imaginary parts to a gr_complex array
59              * \param out_data      pointer to a part of memory of at least nr_elems
60              *                      elements
61              * \param raw_out_data  data that will be converted; pointer to a chunk
62              *                      of memory of at least 2 * nr_elems elements
63              * \param nr_elements  how many complex elements are "prepared"
64              */
65             void prepareOutDataConnex(
66                 gr_complex *out_data,
67                 const int32_t *raw_out_data,
68                 const int nr_elems);
69
70             /* \brief multiplies the line arrays and the column arrays in a chunk
71              */
72             void multLineCol(
73                 std::vector<gr_complex> result,
74                 const gr_complex *line_arr,
75                 const gr_complex *col_arr,
76                 const int nr_chunk);
77
78         public:

```

```

79     multiply_cc_impl(
80         std::string distributionFIFO ,
81         std::string reductionFIFO ,
82         std::string writeFIFO ,
83         std::string readFIFO);
84     ~multiply_cc_impl();
85
86     // Where all the action really happens
87     int work(int noutput_items ,
88             gr_vector_const_void_star &input_items ,
89             gr_vector_void_star &output_items);
90 };
91 } // namespace opincaa
92 } // namespace gr
93
94 #endif /* INCLUDED_OPINCAA_MULTIPLY_CC_IMPL_H */

```

Listing 4: The C++ header for Listing 3

Bibliografie

- [1] Phase Synchronization Capability of TwinRX Daughterboards and DoA Estimation, Ettus Research
- [2] <http://arma.sourceforge.net/>
- [3] <http://www.netlib.org/blas/>
- [4] <https://www.gnu.org/software/octave/>