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).

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

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

Both $\boldsymbol{a}(\theta)$ and $\boldsymbol{V}_{N_{sq}}$ are matrices of complex elements. The real and imaginary parts of $\boldsymbol{a}(\theta)$ are each in the $[-1, 1]$ interval. As of $\boldsymbol{V}_{N_{sq}}$, 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, $\boldsymbol{a}^H(\theta)\boldsymbol{V}_{N_{sq}}$, 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, $X_{1,N}\boldsymbol{a}(\theta)$, 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.

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

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

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

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

The result can be further written as following:

$$X_{1,N} = \begin{bmatrix} \sum_{i=0}^{M-1} (x_i x_{i,0} - y_i y_{i,0}) + j \sum_{i=0}^{M-1} (x_i y_{i,0} + y_i x_{i,0}) \\ \sum_{i=0}^{M-1} (x_i x_{i,1} - y_i y_{i,1}) + j \sum_{i=0}^{M-1} (x_i y_{i,1} + y_i x_{i,1}) \\ \dots \\ \sum_{i=0}^{M-1} (x_i x_{i,M-1} - y_i y_{i,M-1}) + j \sum_{i=0}^{M-1} (x_i y_{i,M-1} + y_i x_{i,M-1}) \end{bmatrix}^T \tag{6}$$

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	PE3	...	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	$x_0x_{0,0}$	$x_1y_{0,0}$	$x_1x_{1,0}$	$y_1y_{1,0}$	...	$x_{M-1}x_{M-1,0}$
R4	$x_0y_{0,0}$		$x_1y_{1,0}$		...	$x_{M-1}y_{M-1,0}$
R5		$y_0x_{0,0}$		$y_1x_{1,0}$	...	

Figura 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.

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

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.

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

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

```
void multiply_kernel(int process_at_once, int size_of_block, int
    blocks_to_reduce)
{
  BEGIN_KERNEL("multiply_arr_mat");
    EXECUTE_IN_ALL(
      R25 = 0;
      R26 = 511;
      R30 = 1;
      R31 = 0;
      R28 = size_of_block;
    )

    EXECUTE_IN_ALL(
      R1 = LS[R25];           // z1 = a1 + j * b1
      R2 = LS[R26];           // z2 = a2 + j * b2
      R29 = INDEX;            // Used later to select PEs for reduction
      R27 = size_of_block;    // Used to select blocks of ARR_SIZE_C for reduction

      R3 = R1 * R2;           // a1 * a2, b1 * b2
      R3 = MULT_HIGH();

      CELL_SHL(R2, R30);      // Bring b2 to the left to calc b2 * a1
      NOP;
      R4 = SHIFT_REG;
      R4 = R1 * R4;           // a1 * b2
      R4 = MULT_HIGH();

      CELL_SHR(R2, R30);
      NOP;
      R5 = SHIFT_REG;
      R5 = R1 * R5;           // b1 * a2
      R5 = MULT_HIGH();

      R9 = INDEX;             // Select only the odd PEs
      R9 = R9 & R30;
      R7 = (R9 == R30);
    )

    EXECUTE_WHERE_EQ(         // Only in the odd PEs
      // Getting -b1 * b2 in each odd cell
      R3 = R31 - R3;          // All partial real parts are in R3

      R4 = R5;                // All partial imaginary parts are now in R4
    )
```

```

    REPEAT_X_TIMES(blocks_to_reduce);
      EXECUTE_IN_ALL(
        R7 = (R29 < R27);     // Select only blocks of 8 PEs at a time by
                              // checking that the index is < k * 8
      )
      EXECUTE_WHERE_LT(
        R29 = 129;            // A random number > 128 so these PEs won't be
                              // selected again
        REDUCE(R3);           // Real part
        REDUCE(R4);           // Imaginary part
      )
      EXECUTE_IN_ALL(
        R27 = R27 + R28;      // Go to the next block of 8 PEs
      )
    END_REPEAT;
  END_KERNEL("multiply_arr_mat");
}
```

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 betweem 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 $\boldsymbol{V}_{N_{sq}}$ 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 corespondence 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.

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 $\boldsymbol{X}_{1xN}$, but the final results needs an additional multiplication of this result by the steering vector $\boldsymbol{a}(\theta)$, and this could also be computed while waiting for the results on the SIMD processor.

Listing 2 presents a pseudocode that explains the work flow.

```

prepare_current data();
for all chunks:
  launch_kernel_execution();
  if not last chunk:
    prepare_next_data();
  if not first chunk:
    process_past_data();
  join_threads();
  read_results();
  increment_pointers();
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 cand be found in Listing 3 in Section §5.

2.4 Integrating the kernel in the MUSIC DoA chain

2.4.1 Adaptations needed for integration

The kernel that multiplies a row vector with a matrix has to be integrated in the **MUSIC Linear Array** block that computes the MUSIC spectrum. Inside this block, the library Armadillo is necessary to compute the eigenvector decomposition of the autocorrelation matrix, so it is better to work with data types specific to Armadillo. An important characteristic of its matrix data types when considering performance is that they are stored in a column-major order. This means that elements of a column are contiguous in memory, so it is desirable to iterate through the columns first and then through the rows for a faster access to data.

The first change, therefore, for integrating the kernel into this block, was to replace that C++ native vectors of complex elements with the Armadillo `fvec` type and use complex matrices of float elements `cx_fmat` where possible, which makes for an easier data management. Wherever possible, the data is passed by references to the functions preparing and processing the data, to avoid unnecessary copying.

2.4.2 Results obtained

The main caveat of using the ConnexArray processor is the conversion from floating point representation to a fixed point one. This conversion will inevitably affect precision and it is necessary to assess its effect on the results. We found that the denomination of the MUSIC spectrum has, on average, an error of approximately 10^{-4}, with a maximum error of around $7 \cdot 10^{-4}$. The problem lies in the fact that, in theory, the value of the denominator is very close to zero for the angle of arrival and in practice it has a value of the order of 10^{-5} up to 10^{-6}, which means that the limitations in the acheivable precision will affect precisely the points of interest.

In practice, we found that for a length of the spectrum of 1024 elements, which translates into an angle resolution of approximately $0,1758°$, there are several points around the angle of interest that are affected by the insufficient precision. In most cases this means that instead of a clear peak around a certain angle, there may be two close peaks which will be designated as the angles of arrival, thus ignoring the correct peak with the lower amplitude of the MUSIC spectrum.

A solution to this problem would be decreasing the length of the spectrum, but which will affect the angle resolution. We have found that when decreasing the length to 512 elements, when the two signal sources are not very close to each other, we obtain an accuracy of the angles of arrival of 0.1°. For a length of 256, in the same scenario, the accuracy decreases to 0.5 degrees.

We are also interested in how close the two sources can be and the angles of arrival to still be distinguished. When the container of the MUSIC spectrum has 512 elements, we obtain correct results with a precision of 0.5° only when the sources are as close as 20° to each other. For a length of the MUSIC spectrum of 256 elements, the sources can be as far as 5° one from each other and the accuracy of the results is of around 0.8°.

Another solution implies changes in the algorithm that finds the local maximum. We could introduce a threshold value such as when the algorithm finds two very close angles (less than one degree apart) it discards one of them and chooses another that is sufficiently distances from the former. This could eliminate the "false positives" while not affecting the overall resolution, but with an additional overhead in finding the local maximum.

2.4.3 Peformance

3 Accelerating the computation of the autocorrelation

The Autocorrelation block has been described in Report 2, Section 4.3. The key concept is that the autocorrelation is estimated through a sample correlation of the signals that arrive at the N antennas in the array, performed over a snapshot period of K samples, after which a Forward-Backward Averaging is computed, which helps the accuracy of the estimation.

Therefore, the first step is computing the sample correlation matrix Cx from the matrix formed by the K samples from the N signals arriving at the antennas.

$$C_x = \frac{1}{K}\boldsymbol{X_K}\boldsymbol{X_K^H}. \tag{9}$$

We observe that $\boldsymbol{C}_X$ is a Hermitian matrix, so $c_{ij} = \overline{c_{ji}}$, where c_{ij} is an element of the matrix and $i = \overline{0, N-1}$, $j = \overline{0, N-1}$. This means that it will suffice to calculate only the elements above and including the main diagonal.

3.1 Autocorrelation kernel on the ConnexArray SIMD

Computing the autocorrelation is reduced to a matrix multiplication which can be offloaded to the ConnexArray. Because we are dealing with large matrices, it is preferable to not perform the whole matrix multiplication in a single ConnexArray job, but instead calculate an element of $\boldsymbol{C}_X$ at a time.

$$\boldsymbol{X}_N \triangleq \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,K-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,K-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1,0} & x_{N-1,1} & \cdots & x_{N-1,K-1} \end{bmatrix} \tag{10}$$

$$\boldsymbol{C}_X = \boldsymbol{X}_N\boldsymbol{X}_N^H = \begin{bmatrix} \sum\limits_{n=\overline{0,K-1}} x_{0,n}x_{n,0}^* & \sum\limits_{n=\overline{0,K-1}} x_{0,n}x_{n,1}^* & \cdots & \sum\limits_{n=\overline{0,K-1}} x_{0,n}x_{n,N-1}^* \\ \sum\limits_{n=\overline{0,K-1}} x_{1,n}x_{n,0}^* & \sum\limits_{n=\overline{0,K-1}} x_{1,n}x_{n,1}^* & \cdots & \sum\limits_{n=\overline{0,K-1}} x_{1,n}x_{n,N-1}^* \\ \vdots & \vdots & \ddots & \vdots \\ \sum\limits_{n=\overline{0,K-1}} x_{N-1,n}x_{n,0}^* & \sum\limits_{n=\overline{0,K-1}} x_{N-1,n}x_{n,1}^* & \cdots & \sum\limits_{n=\overline{0,K-1}} x_{N-1,n}x_{n,N-1}^* \end{bmatrix} \tag{11}$$

$$x_{m,n} = a_{m,n} + jb_{m,n}, \quad m = \overline{0, M-1}, n = \overline{0, M-1} \tag{12}$$

So an element $c_{m,n}$ of $\boldsymbol{C}_X$ is

$$c_{m,n} = \sum_{k=\overline{0,K-1}} (a_{m,k}a_{k,n} + b_{m,k}b_{k,n}) + j\sum_{k=\overline{0,K-1}} (-a_{m,k}b_{k,n} + b_{m,k}a_{k,n}) \tag{13}$$

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~~ Find what's wrong with the results on 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. Autocorrelation

   (a) Make kernel more efficient (think about how to process/prepare data so as to not wait too much).

   (b) How expensive is the writeDataToArray? Is it okay to load everything in one chunk?

   (c) How to do the index loading more efficiently? If it isn't possible to give the line and column indices as arguments to a function, maybe create a big array with possible indices and just send some parts to the kernel? This instead of the current version, which at each iteration deletes the current items and allocates memory for the new index.

   (d) Do the backward averaging part on the kernel too?

   (e) Integrate into gr-doa

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

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "multiply_cc_impl.h"
#include <cmath>
#include <thread>

namespace gr {
  namespace opincaa {

    /*
     * Kernel related functions
     */
    void executeMultiplyArrMat(ConnexMachine *connex);
    void multiply_kernel(
      int process_at_once,
      int size_of_block,
      int blocks_to_reduce);
    void multiply_once(int size_of_block, int blocks_to_reduce);

    multiply_cc::sptr
    multiply_cc::make(
      std::string distributionFIFO,
      std::string reductionFIFO,
      std::string writeFIFO,
      std::string readFIFO)
    {
      return gnuradio::get_initial_sptr
        (new multiply_cc_impl(distributionFIFO,
                              reductionFIFO,
                              writeFIFO,
                              readFIFO));
    }

    /*
     * The private constructor
     */
    multiply_cc_impl::multiply_cc_impl(
      std::string distributionFIFO,
      std::string reductionFIFO,
      std::string writeFIFO,
      std::string readFIFO)
      : gr::sync_block("multiply_cc",
              gr::io_signature::make(2, 2, sizeof(gr_complex) * MAT_SIZE),
              gr::io_signature::make(1, 1, sizeof(gr_complex) * NR_ARRAYS_ELEMS)
                 )
    {
      try {
        connex = new ConnexMachine(distributionFIFO,
                                   reductionFIFO,
                                   writeFIFO,
                                   readFIFO);
      } catch (std::string err) {
        std::cout << err << std::endl;
```

```
    }

    const int blocks_to_reduce = VECTOR_ARRAY_SIZE / ARR_SIZE_C;
    const int size_of_block = ARR_SIZE_C;

    factor_mult1 = 1 << 14;
    factor_mult2 = 1 << 16;
    factor_res = 1 << 14;

    multiply_kernel(PROCESS_AT_ONCE, size_of_block, blocks_to_reduce);
  }

  /*
   * Our virtual destructor.
   */
  multiply_cc_impl::~multiply_cc_impl()
  {
    delete connex;
  }

  void multiply_cc_impl::forecast(
    int noutput_items,
    gr_vector_int &ninput_items_required)
  {
    // The first input is an array of ARR_SIZE elements, while the second is
    // a linearized, column by column matrix of ARR_SIZE x ARR_SIZE elements
    // The result is a 4 array element representing the multiplication between
    // the input array and input matrix.
    ninput_items_required[0] = (NR_ARRAYS / MAT_SIZE) * noutput_items;
    ninput_items_required[1] = noutput_items;
  }

  int
  multiply_cc_impl::work(int noutput_items,
      gr_vector_const_void_star &input_items,
      gr_vector_void_star &output_items)
  {
    if (!connex) {
      return noutput_items;
    }

    const gr_complex *in0 = reinterpret_cast<const gr_complex *>(input_items[0]);
    const gr_complex *in1 = reinterpret_cast<const gr_complex *>(input_items[1]);
    gr_complex *out = reinterpret_cast<gr_complex *>(output_items[0]);

    std::cout << "Called worker with " << noutput_items << " out items." << std::endl;

    for (int i = 0; i < noutput_items; i++) {
      std::cout << "Output item nr. " << i << std::endl;
      const gr_complex *in0_round = &in0[i * NR_ARRAYS_ELEMS];
      const gr_complex *in1_round = &in1[i * MAT_SIZE];
      gr_complex *out_round = &out[i * NR_ARRAYS_ELEMS];

      std::vector<gr_complex> final_res(NR_ARRAYS, 0);

      const int nr_chunks = NR_ARRAYS / ARR_IN_CHUNK;
      const int nr_elem_chunk = ARR_IN_CHUNK * ARR_SIZE;
```

```
    const int nr_elems_calc = PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE / ARR_SIZE_C;

    uint16_t *in0_i = (uint16_t *)
        malloc(nr_chunks * VECTOR_ARRAY_SIZE * sizeof(uint16_t));
    uint16_t *in1_i = (uint16_t *)
        malloc(VECTOR_ARRAY_SIZE * sizeof(uint16_t));
    int32_t *res_mult = (int32_t *)
          malloc(nr_chunks * VECTOR_ARRAY_SIZE * sizeof(int32_t));

    if ((in0_i == NULL) || (in1_i == NULL) || (res_mult == NULL))
      std::cout << "Malloc error!" << std::endl;

    uint16_t *in0_curr, *in1_curr, *in0_next;

    // Prepare the first chunk
    in0_curr = in0_i;
    in1_curr = in1_i;

    int32_t *res_curr_chunk, *res_past_chunk = NULL;
    const gr_complex *arr_next_chunk, *arr_past_chunk;

    const gr_complex *arr_curr_chunk = in0_round;
    const gr_complex *mat = in1_round; // Using the same mat for all chunks
    gr_complex *out_curr_chunk = out_round, *out_past_chunk;

    const int elems_to_prepare = PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE / 2;

    prepareInArrConnex(in0_curr, arr_curr_chunk, elems_to_prepare);
    prepareInMatConnex(in1_curr, mat, elems_to_prepare);

    for (int cnt_chunk = 0; cnt_chunk < nr_chunks; cnt_chunk++) {
      res_curr_chunk = &res_mult[cnt_chunk * VECTOR_ARRAY_SIZE];

      connex->writeDataToArray(in0_curr, PROCESS_AT_ONCE, 0);
      connex->writeDataToArray(in1_curr, PROCESS_AT_ONCE, 511);

      std::thread t(executeMultiplyArrMat, connex);

      try {
        // Prepare future data for all but the last chunk
        if (cnt_chunk != nr_chunks - 1) {
          in0_next = in0_curr + VECTOR_ARRAY_SIZE;
          arr_next_chunk = arr_curr_chunk + nr_elem_chunk;

          prepareInArrConnex(in0_next, arr_next_chunk, elems_to_prepare);
        }

        // Process past data for all but the first chunk
        if (cnt_chunk != 0) {
          out_past_chunk = &out_round[(cnt_chunk - 1) * nr_elem_chunk];
          prepareOutDataConnex(out_past_chunk, res_past_chunk, nr_elems_calc);
          multLineCol(final_res, out_past_chunk, arr_past_chunk, cnt_chunk - 1);
        }
      } catch (...) {
        t.join();
        throw;
      }
```

```
      // Must join threads before reading the reduction
      t.join();

      // 2 * --> complex elements, so we have real *and* imag parts
      connex->readMultiReduction(2 * nr_elems_calc, res_curr_chunk);

      // Incrementing the pointers for the next chunk
      in0_curr = in0_next;
      arr_past_chunk = arr_curr_chunk;
      arr_curr_chunk = arr_next_chunk;
      res_past_chunk = res_curr_chunk;
    } // end loop for each chunk

    // Results for the last chunk
    out_past_chunk = &out_round[(nr_chunks - 1) * nr_elem_chunk];
    prepareOutDataConnex(out_past_chunk, res_past_chunk, nr_elems_calc);
    multLineCol(final_res, out_past_chunk, arr_past_chunk, nr_chunks - 1);

    free(in0_i);
    free(in1_i);
    free(res_mult);
  } // end loop for each output item

  return noutput_items;
}

void multiply_cc_impl::prepareInArrConnex(
      uint16_t *out_arr, const gr_complex *in_arr, const int nr_elems)
{
  for (int j = 0; j < nr_elems; j++) {
    out_arr[2 * j] = static_cast<uint16_t>
      (in_arr[(j / 16) * ARR_SIZE + (j % 4)].real() * factor_mult1);
    out_arr[2 * j + 1] = static_cast<uint16_t>
      (in_arr[(j / 16) * ARR_SIZE + (j % 4)].imag() * factor_mult1);
  }
}

void multiply_cc_impl::prepareInMatConnex(
      uint16_t *out_mat, const gr_complex *in_mat, const int nr_elems)
{
  for (int j = 0; j < nr_elems; j++) {
    out_mat[2 * j] = static_cast<uint16_t>
      (in_mat[j % 16].real() * factor_mult2);
    out_mat[2 * j + 1] = static_cast<uint16_t>
      (in_mat[j % 16].imag() * factor_mult2);
  }
}

void multiply_cc_impl::prepareOutDataConnex(
      gr_complex *out_data,
      const int32_t *raw_out_data,
      const int nr_elems)
{
  float temp0, temp1;
  for (int j = 0; j < nr_elems; j++) {
    temp0 = (static_cast<float>(raw_out_data[j * 2])) / factor_res;
    temp1 = (static_cast<float>(raw_out_data[j * 2 + 1])) / factor_res;

    out_data[j] = std::complex<float>(temp0, temp1);
```

```
    }
  }

  void multiply_cc_impl::multLineCol(
       std::vector<gr_complex> result,
       const gr_complex *line_arr,
       const gr_complex *col_arr,
       const int nr_chunk)
  {
    // Do the final line array * column array multiplication
    // As a result, we have an element for each array processed in a chunk
    std::complex<float> acc(0, 0);
    for (int j = 0; j < ARR_IN_CHUNK; j++) {
      acc = (0, 0);
      for (int jj = 0; jj < ARR_SIZE; jj++) {
        acc = acc +
          (line_arr[j * ARR_IN_CHUNK + jj] * col_arr[j * ARR_IN_CHUNK + jj]);
      }
      result[nr_chunk * ARR_IN_CHUNK + j] = acc;
      std::cout << "result[" << nr_chunk * ARR_IN_CHUNK + j << "] = "
        << result[nr_chunk * ARR_IN_CHUNK + j] << std::endl;
    }
  }

  /*=========================================================================
   * Define ConnexArray kernels that will be used in the worker
   *========================================================================*/
  void executeMultiplyArrMat(ConnexMachine *connex)
  {
    connex->executeKernel("multiply_arr_mat");
  }

  void multiply_kernel(int process_at_once, int size_of_block, int
      blocks_to_reduce)
  {
    BEGIN_KERNEL("multiply_arr_mat");
      EXECUTE_IN_ALL(
        R25 = 0;
        R26 = 511;
        R30 = 1;
        R31 = 0;
        R28 = size_of_block;   // Equal to ARR_SIZE_C; dimension of the blocks
                               // on which reduction is performed at once
      )

      EXECUTE_IN_ALL(
        R1 = LS[R25];            // z1 = a1 + j * b1
        R2 = LS[R26];            // z2 = a2 + j * b2
        R29 = INDEX;           // Used later to select PEs for reduction
        R27 = size_of_block;   // Used to select blocks of ARR_SIZE_C for
            reduction

        R3 = R1 * R2;          // a1 * a2, b1 * b2
        R3 = MULT_HIGH();

        CELL_SHL(R2, R30);     // Bring b2 to the left to calc b2 * a1
        NOP;
        R4 = SHIFT_REG;
        R4 = R1 * R4;          // a1 * b2
        R4 = MULT_HIGH();
```

```
          CELL_SHR(R2, R30);
          NOP;
          R5 = SHIFT_REG;
          R5 = R1 * R5;           // b1 * a2
          R5 = MULT_HIGH();

          R9 = INDEX;             // Select only the odd PEs
          R9 = R9 & R30;
          R7 = (R9 == R30);
        )

        EXECUTE_WHERE_EQ(         // Only in the odd PEs
          // Getting -b1 * b2 in each odd cell
          R3 = R31 - R3;          // All partial real parts are in R3

          R4 = R5;                // All partial imaginary parts are now in R4
        )

        REPEAT_X_TIMES(blocks_to_reduce);
          EXECUTE_IN_ALL(
            R7 = (R29 < R27);     // Select only blocks of 8 PEs at a time by
                                  // checking that the index is < k * 8
          )
          EXECUTE_WHERE_LT(
            R29 = 129;            // A random number > 128 so these PEs won't be
                                  // selected again
            REDUCE(R3);           // Real part
            REDUCE(R4);           // Imaginary part
          )
          EXECUTE_IN_ALL(
            R27 = R27 + R28;      // Go to the next block of 8 PEs
          )
        END_REPEAT;

      END_KERNEL("multiply_arr_mat");
    }
  } /* namespace opincaa */
} /* 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

```
#ifndef INCLUDED_OPINCAA_MULTIPLY_CC_IMPL_H
#define INCLUDED_OPINCAA_MULTIPLY_CC_IMPL_H

#include <opincaa/multiply_cc.h>
#include "ConnexMachine.h"

#define VECTOR_ARRAY_SIZE 128
#define ARR_SIZE 4
#define MAT_SIZE (ARR_SIZE * ARR_SIZE)

#define ARR_SIZE_C (ARR_SIZE * 2)
#define MAT_SIZE_C (MAT_SIZE * 2)

#define PROCESS_AT_ONCE 1

// Defining a cluster of 16 arrays of 4 elements each that are multiplied with
// the same matrix
#define NR_ARRAYS 32
```

```
#define NR_ARRAYS_ELEMS (NR_ARRAYS * ARR_SIZE)

// how many arr * mat multiplications can be performed on the ConnexArray in a
// round aka. in a chunk
#define ARR_IN_CHUNK (PROCESS_AT_ONCE * VECTOR_ARRAY_SIZE / MAT_SIZE_C)

#define MIN(x, y) ((x > y) ? y : x)


namespace gr {
  namespace opincaa {

    class multiply_cc_impl : public multiply_cc
    {
     private:
      ConnexMachine *connex;

      // Factors required for scaling the data
      int factor_mult1;
      int factor_mult2;
      int factor_res;

      void forecast(
        int noutput_items,
        gr_vector_int &ninput_items_required);

      /* \brief scales and casts a number of nr_elems input elements in an array
       *        of size ARR_SIZE
       */
      void prepareInArrConnex(
        uint16_t *out_arr, const gr_complex *in_arr, const int nr_elems);

      /* \brief scales and casts a number of nr_elems input elements in an
       *        matrix of size MAT_SIZE x MAT_SIZE
       */
      void prepareInMatConnex(
        uint16_t *out_mat, const gr_complex *in_mat, const int nr_elems);

      /* \brief Scales back the output data and converts the array of
       *        real and imaginary parts to a gr_complex array
       * \param out_data         pointer to a part of memory of at least nr_elems
       *                         elements
       * \param raw_out_data     data that will be coverted; pointer to a chunk
       *                         of memory of at least 2 * nr_elems elements
       * \param nr_elements      how many complex elements are "prepared"
       */
      void prepareOutDataConnex(
        gr_complex *out_data,
        const int32_t *raw_out_data,
        const int nr_elems);

      /* \brief multiplies the line arrays and the column arrays in a chunk
       */
      void multLineCol(
        std::vector<gr_complex> result,
        const gr_complex *line_arr,
        const gr_complex *col_arr,
        const int nr_chunk);

     public:
```

```
      multiply_cc_impl(
        std::string distributionFIFO,
        std::string reductionFIFO,
        std::string writeFIFO,
        std::string readFIFO);
      ~multiply_cc_impl();

      // Where all the action really happens
      int work(int noutput_items,
         gr_vector_const_void_star &input_items,
         gr_vector_void_star &output_items);
    };
  } // namespace opincaa
} // namespace gr

#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/