HPPS Project: RTM - Maxeler

Enrico Deiana & Emanuele Del Sozzo

June 23, 2013

Contents

1 Abstract

The objective of this project is to accelerate in hardware the RTM application using the Maxeler platform. First of all, it was necessary to analyze the C code (part 2) to find out the critical parts. This analysis was done both by looking at the C code and by using profiling tools (part 3). Meanwhile, we started to study how the Maxeler platform works by reading the manuals and looking at the examples inside Maxide. Then, we decided which function was the most candidate to be implemented in hardware. During the implementation, we followed several approaches (part 4), from trivial to more complex ones. In the end, only with the last approach we were able to synthetize the kernel on the FPGA board and examine the result of hardware acceleration. In part 5 we explain which are the possible enhancements and the comparison between the various implementations of RTM (with pragma omp library, without it, and our final approach).

2 RTM

Reverse Time Migration (RTM) is the current state-of-the-art in seismic imaging. It aims at constructing an image of the subsurface from recordings of seismic reflections.

It is mainly made up of two processes: 1) seismic forward modelling of a wave field originating from the source position, 2) reverse-time modelling of the shot-record, where the time samples are put into the modelling scheme.

The idea behind this procedure is that sufficiently strong correlation amplitudes can only be expected at subsurface locations where a reflection occurs. Occurrence of a reflection at a certain location means that there is a down-going wave field from the source side and an up-going wave field towards the receivers at the same time.

The implementation of RTM that has been provided to us is composed by three C source files *cpu_main.c*, *cpu_constructors.c*, *cpu_simulation.c* and two header files *cpu_constructors.h* and *cpu_simulation.h* which declare the functions present in their respective source file.

cpu_main.c contains just the main function which is in charge of calling all the subroutines that: initialize the random seed for the random field creation and all the needed parameter, construct the model and all the necessary structures for the computation, simulate the model and store the results.

cpu_constructors.c implements all the functions responsible for allocation and initialization of all global variables and required data structures. Parsing of the parameter file, earth model file when required, and receiver input data file when available. Construction of structures required during simulation.

cpu_simulation.c includes all the functions responsible for wave equation, injection of the source signal, saving the acquired receiver signal, propagating the source/receiver, removing the source signal from the receiver acquisition data, correlating the wavefields to create subsurface images, reinjecting the receiver data during back propagation, backpropagating both the receiver and the source and correlating the wavefields.

Several files containing the parameters needed for the creation of the model and for the simulation are provided in the *tests* directory. These are: *rtm_parameters_small.txt*, *rtm_parameters_medium.txt*, *rtm_parameters_large.txt*, *marmousi_0.txt*, *marmousi_32.txt* . For our tests we have used the small one. More informations on the structure of the program are provided into the README.txt file.

As we will see in the profiling section, what we are particulary intrested in are two functions in which the program spends most of its time: the *do_step* and *do_step_damping* which are in the *cpu_simulation.c* file and are in charge of propagating the wavefield.

For a better comprehension of the program execution flow we state here a very simple version of RTM in pseudo C code, containing just the interesting functions that we found out looking at the code and during the profiling.

Here is the *main*:

```
1  main(){
2
3    create_params_fields_data();
4
5    if(!RECEIVER_FILE){
6      prop_source();
```

```
7       CREATION_RECEIVER_FILE();
8     }
9
10    prop_source();
11
12    migrate_shot();
13
14    dump_image_to_file();
15    clean();
16
17  }
```

It takes as single argument the *rtm_parameters_[small/medium/large].txt* file containing all the needed parameters and data for the model creation. The function *create_params_fields_data()* creates the needed structures for the following computational steps (in the real RTM code there are different functions that do it, but they are done sequentially and are computationally meaningless). In case there is not the *RECEIVER_FILE* (that is a file created during the very first execution of the application, containing some values and parameters) is called *prop_source()* that do some computation and then the *RECEIVER_FILE* is created. The remaining part of the code is executed always in the same way. The functions *prop_source()* and *migrate_shot()* are called and do the most of the computational part of the program, then *dump_image_to_file()* saves the results and *clean()* clears the program state.

As you will see in the profiling section (the next one) *prop_source()* and *migrate_shot()* are in charge of calling *do_step_damping()* and *do_step()* which do most of the computational effort.

Here is the *prop_source()*:

```
1   prop_source(){
2
3     for(i=0; i<NT; i++){
4
5       add_source(SOURCE_CONTAINER);
6
7       if(!RECEIVER_FILE){
8         do_step_damping(P, PP, SOURCE_CONTAINER);
9         extract_data();
10      }
11      else{
12        do_step(P, PP, SOURCE_CONTAINER);
13      }
14
15      FLIP_P_PP();
16
17    }
18
19  }
```

The function *add_source()* is not computational interesting, but it modifies *SOURCE_CONTAINER* that is passed to *do_step()* and *do_step_damping()*. In case it is the first program execution (!RECEIVER_FILE is true) *prop_source()* will be executed the first time calling *NT* times *do_step_damping()* and *extract_data()* (that is not very interesting from a computational point of view), and the second time calling *do_step()* for *NT* times again. At the end *P* and *PP*, that are arguments of *do_step()* and *do_step_damping()*, are exchanged (*FLIP_P_PP()* is not a function present in the real RTM code which just flips the pointers). If it is a subsequent program execution, just the second part is done.

Here is the *migrate_shot()*:

```
1   migrate_shot(){
2
3     for(i=0; i<NT; i++){
4
5       clean_source();
6       do_step(P, PP, SOURCE_CONTAINER);
7       add_data();
8       do_step_damping(P, PP, SOURCE_CONTAINER);
```

```
 9        image_it();
10        FLIP_P_PP();
11
12     }
13
14  }
```

This function is executed anyway whether *RECEIVER_FILE* is present or not. It executes alternatively *NT* times *do_step()* and *do_step_damping()*, our most interesting functions since the program spends a lot of time into them (especially inside *do_step_damping()*) and into *image_it()* that applies the imaging condition.

The result of *do_step()* and *do_step_damping()* is stored in their argument PP. The behavior of these functions is better explained in part 4.

NT is the number of time steps and is defined as $JT \cdot RF$ which are defined into the *rtm_parameters_small.txt*, *rtm_parameters_medium.txt*, *rtm_parameters_large.txt* files passed as single argument to RTM. In the *small* case NT=1000, in the *medium* one NT=2500 and in the *large* one NT=3000.

3 Profiling

First of all we looked at the code to identify the most critical parts of the program. We immediately detected two functions, *do_step* and *do_step_damping*, with a very heavy computation consisting in three nested loops. But for a better application analysis, in order to know how much time the program spends inside its differents subroutines, we profiled the RTM application (with the *rtm_parameters_small.txt* input file and no RECEIVER_FILE) with three well known profiling tools: *callgrind*, *gprof*, *oprofile*. However, the most interesting results are returned by *callgrind* and *gprof*, while *oprofile* results are rather similar to the *callgrind* ones (moreover with some lack of informations).

3.1 Callgrind

Callgrind is part of valgrind's tool suite. It records the call history among functions in a program's run as a call-graph. The main advantage of callgrind is that we don't need to recompile the application, but this tool adds a huge amount of overhead during the program execution (that took approximately 15 times more than the normal execution).

We have profiled the RTM application using the small data:

```
$ valgrind --tool=callgrind rtm-cpu ../../tests/rtm_parameters_small.txt
```

What we found out is that RTM spends most of the time (63.89%) inside *do_step_damping* propagating the wavefield of one step in time each time it is called. The callers are *prop_source* and *migrate_shot*. The first one propagates the source forwards, the second one back propagates the source and the receiver and correlates the wavefields, both call *do_step_damping* 1000 times, so it is called 2000 times overall.

Here is the explanation of graphs:

- Boxes: represent functions. Each one is labeled by the function name and contains a bar representing the time (in percentage wrt the execution time) spent there by the application. Note that the shown cost is only the cost which is spent while the active function was actually running; i.e. the cost shown for *main* should be the same as the cost of the active function, as that is the part of inclusive cost of *main* spent while the active function was running.

- Arrows: represent calls. An arrow starts from the caller and ends into the callee. The bar on its right side indicates the number of times that the child node is invoked by the parent node, and the more is thick the more computational expensive the calls are.

Below is shown the callee graph for this function:

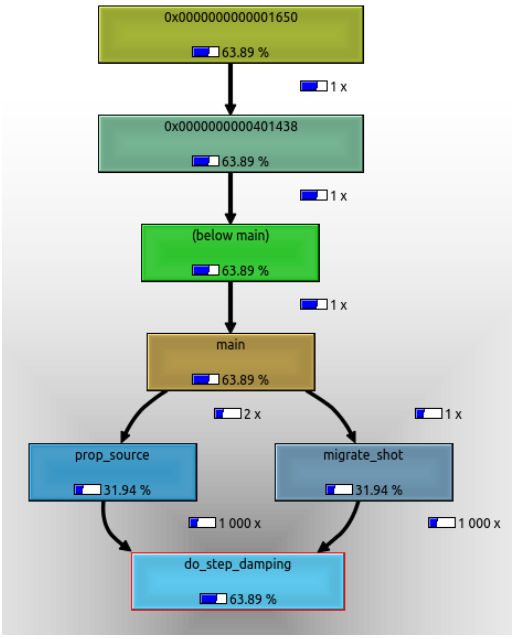

Figure 1: do_step_damping

Another function, very similar to the *do_step_damping*, is *do_step*, where the program spends 15.17% of its time. Even this time the callers are *prop_source* and *migrate_shot* that calls it again 1000 times each. The callee graph is this one:

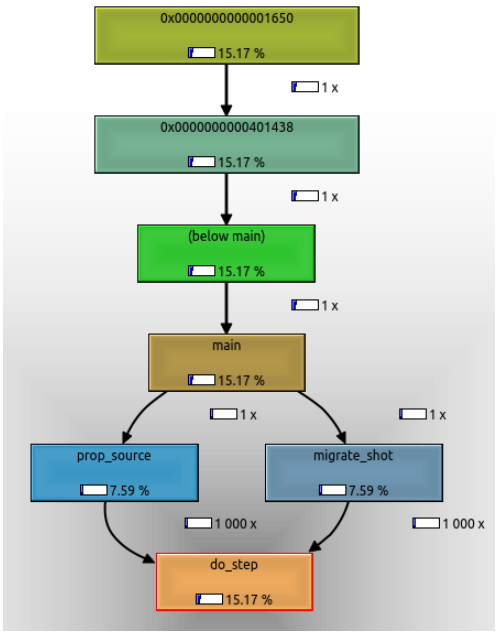

Figure 2: do_step

Finally *image_it* is the last more onerous function, it applies the imaging condition after the backpropagation

of the source and the receiver. RTM spends here, according to callgrind, the 20.22% of its time. The caller is *migrate_shot* that calls *image_it* 1000 times. The callee graph is:

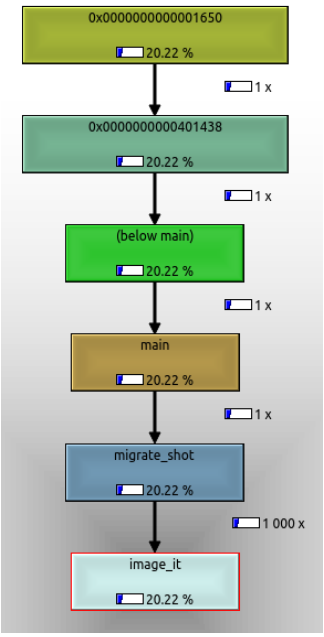

Figure 3: image_it

The overall meaningful calls done by *main* are shown here:

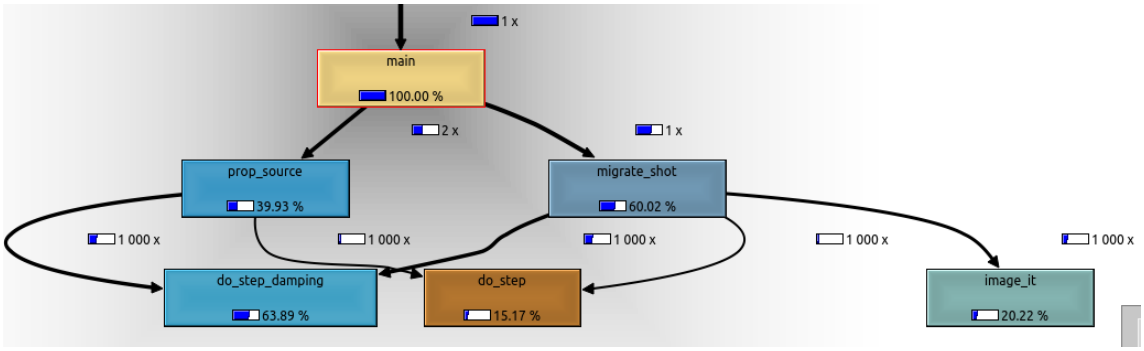

Figure 4: main

3.2 Gprof

Gprof is the GNU profiling tools. It needs to recompile the application for doing the profiling with the *-pg* option, so we have modified the *MakeFile* and built again the program, but the needed overhead for the profiling part is quite small so that it almost doesn't affect the program execution.

Respect to callgrind there are some mismatches, especially for *do_step* and *image_it*, where gprof states that the program spends 17.7% and 16.9% respectively (instead of 15.17% and 20.22% stated by callgrind). While for the *do_step_damping* function the amount of time spent is quite similar (64.3% wrt 63.89%). The number of calls and the caller is, obviously, the same stated before in the callgrind part (2000× *do_step_damping* and *do_step*, 1000× *image_it*).

More informations about this profiling done with gprof can be found in the "analysis.txt" file that reports all the profiling informations (the caller and callee graph, the number of calls of each functions, the spent time etc.).

4 Approaches

According to the profiling, we decided to work on the do_step and do_step_damping functions and to accelerate them in hardware.

Here the code of the do_step function:

```
1  /*
2   * do_step - propagate wavefield one step in time
3   * p  - current wavefield
4   * pp - previous and next wavefield
5   * dvv - density (1) * velocity *velocity
6   */
7  void do_step(float *__restrict p, float *__restrict pp, float *__restrict dvv, float *__restrict
       source_container){
8    int i3;  //Indexes
9    int n12=n1*n2;
10
11 // #pragma omp parallel for
12   for(i3=ORDER; i3 < n3-ORDER; i3++){    //Loop over slowest axis
13     int i1;
14     int i2;
15     for(i2=ORDER; i2 < n2-ORDER; i2++){ //Loop over middle axis
16       for(i1=ORDER; i1< n1-ORDER; i1++){  //Loop over fast axis
17         //Wavefield update
18         pp[i1+i2*n1+i3*n12]=(2.0*p[i1+i2*n1+i3*n12]-pp[i1+i2*n1+i3*n12]+dvv[i1+i2*n1+i3*n12]*(
19             p[i1+i2*n1+i3*n12]*c_0
20             +c_1[0]*(p[(i1+1)+(i2  )*n1+(i3  )*n12]+p[(i1-1)+(i2  )*n1+(i3  )*n12])
21             +c_1[1]*(p[(i1+2)+(i2  )*n1+(i3  )*n12]+p[(i1-2)+(i2  )*n1+(i3  )*n12])
22             +c_1[2]*(p[(i1+3)+(i2  )*n1+(i3  )*n12]+p[(i1-3)+(i2  )*n1+(i3  )*n12])
23             +c_1[3]*(p[(i1+4)+(i2  )*n1+(i3  )*n12]+p[(i1-4)+(i2  )*n1+(i3  )*n12])
24             +c_1[4]*(p[(i1+5)+(i2  )*n1+(i3  )*n12]+p[(i1-5)+(i2  )*n1+(i3  )*n12])
25             +c_2[0]*(p[(i1  )+(i2+1)*n1+(i3  )*n12]+p[(i1  )+(i2-1)*n1+(i3  )*n12])
26             +c_2[1]*(p[(i1  )+(i2+2)*n1+(i3  )*n12]+p[(i1  )+(i2-2)*n1+(i3  )*n12])
27             +c_2[2]*(p[(i1  )+(i2+3)*n1+(i3  )*n12]+p[(i1  )+(i2-3)*n1+(i3  )*n12])
28             +c_2[3]*(p[(i1  )+(i2+4)*n1+(i3  )*n12]+p[(i1  )+(i2-4)*n1+(i3  )*n12])
29             +c_2[4]*(p[(i1  )+(i2+5)*n1+(i3  )*n12]+p[(i1  )+(i2-5)*n1+(i3  )*n12])
30             +c_3[0]*(p[(i1  )+(i2  )*n1+(i3+1)*n12]+p[(i1  )+(i2  )*n1+(i3-1)*n12])
31             +c_3[1]*(p[(i1  )+(i2  )*n1+(i3+2)*n12]+p[(i1  )+(i2  )*n1+(i3-2)*n12])
32             +c_3[2]*(p[(i1  )+(i2  )*n1+(i3+3)*n12]+p[(i1  )+(i2  )*n1+(i3-3)*n12])
33             +c_3[3]*(p[(i1  )+(i2  )*n1+(i3+4)*n12]+p[(i1  )+(i2  )*n1+(i3-4)*n12])
34             +c_3[4]*(p[(i1  )+(i2  )*n1+(i3+5)*n12]+p[(i1  )+(i2  )*n1+(i3-5)*n12])
35             ))+source_container[i1+i2*n1+i3*n12];
36       }
37     }
38   }
39
40   return;
41 }
```

The code of the do_step_damping function is fairly similar, the only difference is that before and after the assignment of a value to the array pp, we have this assignment:

```
pp[i1+i2*n1+i3*n12]*=scale;
```

where the variable scale is computed at each interation of the most internal for.

So, these two sub-routine seemed the most suitable to be translated into maxeler java code since they were a series of operations repeated at each iteration and indipendent from the results just computed in that call, hence they could be easily pipelined by the machine.

4.1 First simple approaches

The first approaches were aimed at getting more confident with the Maxeler SLiC interface, Manager and Kernel. At the beginning, we used the Basic SLiC interface and our first approach consisted in computing, at each iteration of the most internal for, the value of pp on the DFE. The application, to compute each value of the pp array, uses a 3D cross centered in the point at offsets i1+i2*n1+i3*n12. Our idea was to copy, at each iteration, the values of

the cross into an array and pass it as a stream to the DFE, which had simply to sum them and then return the value.

So, we created a stream containing the cross of elements, i.e., considering p[i1+i2*n1+i3*n12], the five previous and next elements of each dimension, and the values of arrays pp, dvv, source_container, c_1, c_2, c_3, c_4 and c_0. This approach, of course, was very trivial since we were using the Maxeler platform only for computing a value at each for iteration, just like a C sub-routine, instead of exploiting the potentiality and the nature of the machine, hence we decided to drop it without sythentizing it. Indeed, we would have spent more time in creating the stream and getting the output, through PCI Express, than computing the value.

Another possible approach, developed by other group working on Maxeler RTM, was passing the arrays p, pp, dvv, source_container as streams to the DFE and using the same offsets that we would use in the C code to compute the output stream (an approach very similar to the C code). However, this was not feasible since the space needed to keep in BRAM (the memory on board, usually sized into kb) all the values required to compute a single output value was to much. Indeed, the more the offsets and the data get bigger the more space on BRAM is required.

So, we moved to two new different approaches: one with LMEM Blocked 3D, the other with Stalling Streams.

4.2 LMEM Blocked 3D

The DFE has two types of memory: FMem (Fast Memory) which can store several megabytes of data on-chip with terabytes/second of access bandwidth and LMem (Large Memory) which can store many gigabytes of data off-chip. In this approach, we used the dynamic SLiC interface to have complete management of the communication between C code and Manager. Our idea was to load in LMEM the array p as a 3D block. In this way, it would be possible to access to all its subcube. As it can be seen by looking at the code of the do_step, at each iteration, the computation of a value of pp required a 3D cross of dimension 11*11*11 (stencil size). In this way, we thought we could access every subcube of p of dimension 11*11*11 and compute the value of pp[i1+i2*n1*i3*n12]. This approach would avoid to fill the BRAM since we would load only the values required by that computation without using big offsets. In addiction, we wanted to create several streams containing only the values of pp, dvv, source_container required to compute the output stream, that would be stored in LMEM. After the computation, we would read the output from LMEM and stored in the pp array. So, we can see this approach as a combination of the previous approaches, in a way. At the beginning, we thought we could move the subcube from the Kernel itself, but this was not possible. The only chance to do this was to pass through SLiC interface the offsets of the starting points of the 3D subcube from the C code. However, we begun to have several problems due to Maxeler machine architecture. In the manual, we found the function used by the Manager to select a subcube of the 3D block and pass it to the Kernel. This is the function:

```
public void setLMemBlocked(String streamName, long address,
long arraySizeFast, long arraySizeMed, long arraySizeSlow, long rwSizeFast, long
    rwSizeMed, long rwSizeSlow, long offsetFast, long offsetMed, long offsetSlow)
```

where the first three parameters are refered to the size 3D block in LMEM, the next three to the size of the subcube we want to consider, the last three are the offsets the subcube starts from.

This function was neither completely explained in the manual nor used in examples, so, before applying it to the RTM application, we modified an example in Maxide and made some tests on it. It turned out that this function wanted at least one of the three size of the 3D block in LMEM multiple of the Maxeler burst, in order to have it burst-aligned. The burst of Maxeler machine is 384 byte, hence we had to load a block of size x*y*z, where at least one of these three parameter had to be multiple of 96 (indeed, we used arrays of 32-bit float, so 4 byte per float). The size of stream p, like the size of streams pp, dvv, source_container, is given by the test files of the RTM application and none of its three dimensions are multiple of 96, thus we had to modify one of these dimensions (the fast one) and make it multiple of 96. However, in this way we added useless data to the streams but it was still possible to extract only the data we needed at the end of the computation. Another problem occured when we found out that not only one of the 3D block dimensions had to be multiple of 96, but also one of the dimensions of the subcube. This fact made our solution more complicated since we couldn't take subcubse of size 11*11*11 anymore, but we had to manage bigger subcubes. A possible solution was to compute from this new subcube as many useful values as we could and then move the offsets from the C code to compute the following values. For instance, if we

were using the rtm_parameters_small.txt, the size of the arrays would be 100*100*100, then it would be modified to 192*100*100. The first subcube would start from offsets [0, 0, 0] and would be of size 96*11*11. The first execution of Kernel would produce 86 useful values (from [5,5,5] to [91,5,5]). Then, we would move the offsets to [92,0,0]. In fact, the only useful values we would need were the values at offsets [93,5,5], [94,5,5], [95,5,5], the others were useless. However, at this point the last and most critical problem rose: even at least one of the offsets had to be multiple of 96. As consequence, we were unable to exploit this approach since it was impossible to access to subcubes starting from offsets x,y,z where none of them was multiple of 96.

4.3 Stalling Streams

Our last approach was about stalling streams. The general idea behind this approach is: we have an array (p) and we access it as it was a 3D array, using three different offsets, so we can be linearized in three other arrays by following the dimensions x, y, z and then use them to compute the values of the output. Of course, the main issue of this approach is to synchronize the three arrays and make them flow asynchronously. In fact, we decided to have only of array linearized in one dimension (z), then move the 3D cross in that direction and collect the elements on it along the other two dimensions in the remaining arrays.

We designed an algorithm based on three streams: the first one was an array linearized by z dimension (p itself), the other two were arrays filled, respectively, with 10 elements in x and y dimensions necessary to compute each value of pp. To do so, we created at each execution of do_step and do_step_damping two new arrays: px and py. Each one of these contained n1*n2*n3*11 elements (to avoid mistakes by skipping the central value already in p). Because of the different size, we needed to find a method to synchronize these two arrays with the array p. Fortunately, Maxeler gives the opportunity to stall not only input streams but also output streams. Therefore, we created a controller of size n1*n2*n3*11 and filled it with blocks of one 1 and ten 0. The idea is: when the input/ouput stream on the Kernel has the 1, it can use/save the actual value and then to the next one. When it has 0, it can't go further. So, the streams p, pp, dvv, source_container would be driven by the controller, instead streams px and py would be free to move on at each tick. Moreover, there is no need to load streams in LMEM since the number of elements required to compute each output value was limited. Indeed, only when we have 1 on the controller we write the value on the output by using the 10 elements of px and py, the 11 elements of p and elements from the other streams, not so much space was required in BRAM. Of course, one of the drawbacks was the fact that both px and py contained duplicated data but it was the simpliest way to create the two streams. Another problem came from Maxeler because it didn't accept that a stream (pp) was both an input and output stream, hence we created a new output stream, ppresult, and then, at the end of the do_step/do_step_damping, we shifted the pointers of ppresult and pp. Moreover, we had to deal with the time of creation of px and py: both were created in three nested for, in total (n1-10)*(n2-10)*(n3-10) iterations (the same number of iterations of the original C code). Our first implementation of this algorithm was quite trivial, since we used the DFE only in the do_step sub-routine. At each call of the do_step we allocated the DFE (100msec to 1sec) and passed the 7 input streams (p, pp, dvv, source_container, px, py, controller), one output stream (ppresult) and several scalar values (c_0, c_1[], c_2[], c_3[], c_4[]). The Kernel received these streams and, when controller was 1, it computed the value of ppresult by using almost the same formula of C code.

Here we can se the Manager, the Kernel, and the do_step sub-routine using SLiC functions:

Manager:

```
1  public class CpuMainManager{
2
3      private static final String s_kernel1 = "linearKernel";
4
5       public static void main(String[] args) {
6
7           /*...*/
8
9           //link of streams for CPU to DFE
10          manager.setIO(
11              link("p", IODestination.CPU),
12                  /*...*/
13              );
14
15         /*...*/
16     }
```

```
17
18      private static EngineInterface interfaceDefault() {
19          EngineInterface engine_interface = new EngineInterface();
20
21            /*...*/
22
23          //creation of InterfaceParam
24
25            InterfaceParam  c_1_0    = engine_interface.addParam("c_1_0", CPUTypes.DOUBLE);
26
27            /*...*/
28
29          //set of Kernel ticks
30
31            engine_interface.setTicks(s_kernel1, sizeController);
32
33            //set of InterfaceParam as streams and scalars
34
35            engine_interface.setStream("p", type, size*sizeFloat);
36            engine_interface.setScalar(s_kernel1, "c_0", c_0);
37
38            /*...*/
39
40            engine_interface.ignoreAll(Direction.IN_OUT);
41          return engine_interface;
42      }
43  }
```

Kernel:

```
1   class CpuMainKernel extends Kernel {
2
3       protected CpuMainKernel(KernelParameters parameters) {
4         super(parameters);
5
6           //creation of DFEVar
7
8           DFEVar controller = io.input("controller", typeInt);
9         DFEVar p = io.input("p", typeFloat, controller.cast(dfeBool()));
10
11          /*...*/
12
13          DFEVar result= 2.0*p-pp+dvv*(
14              p*c_0
15              +c_1_0*(stream.offset(p, 1)+stream.offset(p,-1))
16              +c_1_1*(stream.offset(p, 2)+stream.offset(p,-2))
17              +c_1_2*(stream.offset(p, 3)+stream.offset(p,-3))
18              +c_1_3*(stream.offset(p, 4)+stream.offset(p,-4))
19              +c_1_4*(stream.offset(p, 5)+stream.offset(p,-5))
20              +c_2_0*(stream.offset(py, -4) + stream.offset(py, -6))
21              +c_2_1*(stream.offset(py, -3) + stream.offset(py, -7))
22              +c_2_2*(stream.offset(py, -2) + stream.offset(py, -8))
23              +c_2_3*(stream.offset(py, -1) + stream.offset(py, -9))
24              +c_2_4*(py + stream.offset(py, -10))
25              +c_3_0*(stream.offset(px, -4) + stream.offset(px, -6))
26              +c_3_1*(stream.offset(px, -3) + stream.offset(px, -7))
27              +c_3_2*(stream.offset(px, -2) + stream.offset(px, -8))
28              +c_3_3*(stream.offset(px, -1) + stream.offset(px, -9))
29              +c_3_4*(px + stream.offset(px, -10))
30              )+source_container;
31
32          io.output("ppresult", result, typeFloat, controller.cast(dfeBool()));
33      }
34
35  }
```

do_step:

```
1   void do_step(float *__restrict p, float *__restrict pp, float *__restrict dvv,
2         float *__restrict source_container) {
3
4       //variables creation
5
6       float *px;
7
8       /*...*/
9
10      //several data allocatations
```

```
11
12      uint32_t *controller = malloc(size * stencilSize * sizeof(uint32_t));
13      ppresult = malloc(sizeBytes);
14      px = malloc(sizepxy);
15      py = malloc(sizepxy);
16
17       /*...*/
18
19      for (i3 = ORDER; i3 < n3 - ORDER; i3++) { //Loop over slowest axis
20         for (i2 = ORDER; i2 < n2 - ORDER; i2++) { //Loop over middle axis
21            for (i1 = ORDER; i1 < n1 - ORDER; i1++, index++) {
22
23                  //assignment of values to px and py
24               px[((i1) + (i2) * n1 + (i3) * n12) * stencilSize - 10] = p[(i1)
25                     + (i2) * n1 + (i3 - 5) * n12];
26               py[((i1) + (i2) * n1 + (i3) * n12) * stencilSize - 10] = p[(i1)
27                     + (i2 - 5) * n1 + (i3) * n12];
28
29                  /*...*/
30            }
31         }
32      }
33
34      max_file_t *maxfile = CpuMain_init();
35      max_engine_t *engine = max_load(maxfile, "*");
36
37      max_actions_t* act = max_actions_init(maxfile, "default");
38
39       //loading streams and scalars
40      max_queue_input(act, "p", p, size * sizeof(float));
41      max_queue_output(act, "ppresult", ppresult, size * sizeof(float));
42
43      max_set_param_double(act, "c_0", (double) c_0);
44
45       /*...*/
46
47      //running engine
48      max_run(engine, act);
49      max_unload(engine);
50
51      /*...*/
52
53      return;
54  }
```

After several tests in simulation, we decided to synthetize the Kernel on the FPGA. Once the process was completed, we had these results regarding the resource usage:

Logic utilization:	30518 / 297600	(10,25%)
LUTs:	22879 / 297600	(8,02%)
Primary FFs:	26354 / 297600	(8,86%)
Secondary FFs:	4085 / 297600	(1,37%)
Multipliers (25x18):	34 / 2016	(1,69%)
DSP blocks:	34 / 2016	(1,69%)
Block memory (BRAM18):	59 / 2128	(2,77%)

Table 1: resource usage not-optimized Kernel

Then, we run it to see its performance. It turned out that, using the file *rtm_parameters_small.txt*, the original RTM application run on Intel i7 processor of Maxeler, without pragma, took nearly 2 minutes, while the RTM application accelerated in hardware took 26 minutes. We expected this result since our implementation was not optimized, this run was only useful as proof of the termination of our application. So, we begun to optimized the code by adding/editing several parts of it. Here the list of optimizations we implemented:

- the DFE is allocated only one time, then it's deallocated before the end of the main;
- the do_step_damping sub-routine is accelerated in hardware too. To do so, we created a new array containing the values of variable scale generated during the execution of do_step_damping;

- editing of the kernel to make it common for both do_step and do_step_damping (added the stream scale);
- the arrays px, py, scale, controller are allocated once with the DFE;
- since it never changes, the controller is loaded in the LMEM of the FPGA one time only.

After these changes, we were ready to synthetize this new Kernel, here the results of the resource usage:

Logic utilization:	73339 / 297600	(24.64%)
LUTs:	50815 / 297600	(17.07%)
Primary FFs:	61121 / 297600	(20.54%)
Secondary FFs:	11365 / 297600	(3.82%)
Multipliers (25x18):	28 / 2016	(1.88%)
DSP blocks:	38 / 2016	(1.88%)
Block memory (BRAM18):	252 / 2128	(11.84%)

Table 2: resource usage optimized Kernel

Here the Kernel structure:

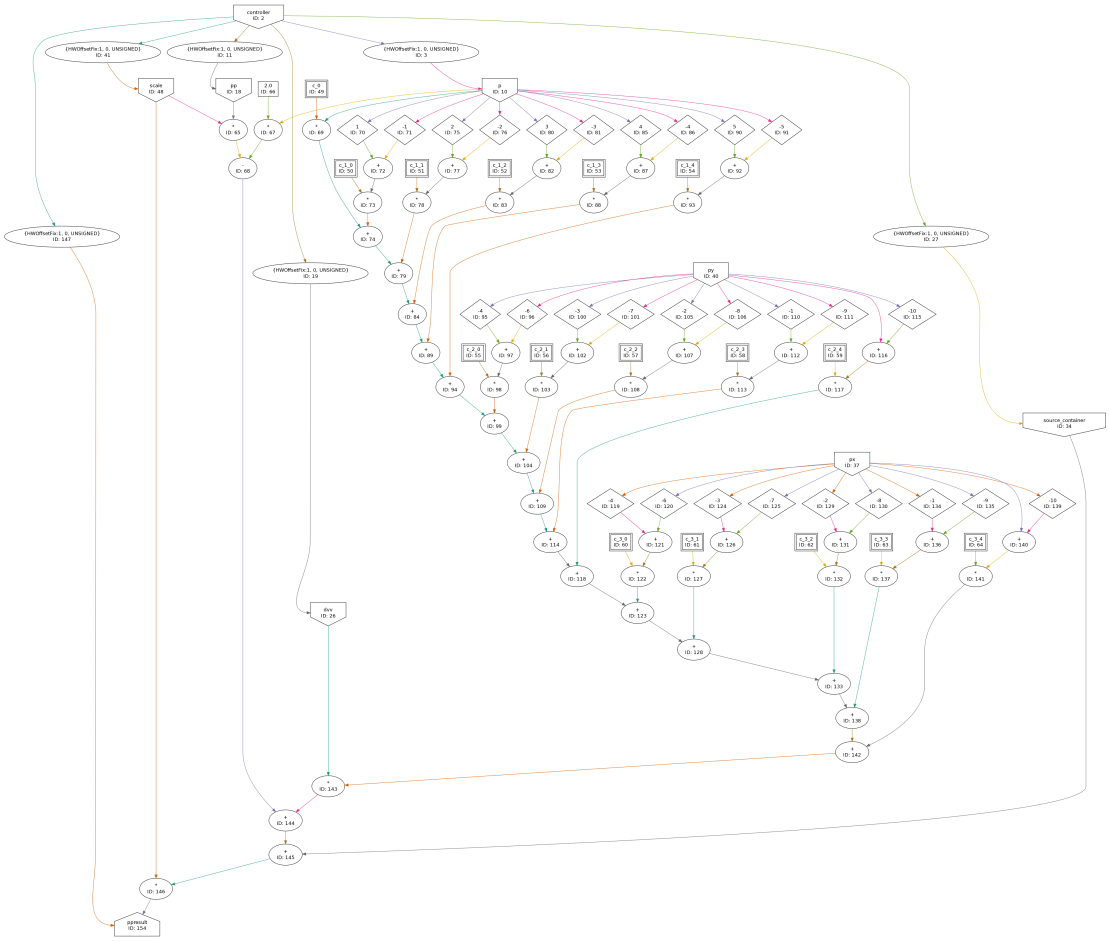

Figure 5: Optimized Kernel original structure

The performance of our application improved but it was still slower than the original RTM application. Indeed, now it took nearly 10 minutes to complete. So, for sure, there were some bottlenecks inside our code. The most probable bottlenecks where two: the preparation of arrays px and py and the stream of data from the RAM of i7 to the FPGA via PCI Express. To find out the problem, we made several tests to analyze the time of preparation the array px/py and the time of streaming data/computing the data on DFE. It turned out that the creation of the arrays took, on average, nearly 0.014s, while the streaming/computation nearly 0.20s. So, the bottleneck came from either the data streaming or the data computation. We very pretty sure the problem was in data streaming, thus we made some tests by creating a very simple Kernel, in another project, that received the same number of input streams, as our RTM Kernel, same size and we put in the output stream the values of one of the input streams, to minimize the computation time. Of course, also these streams were controlled by a controller loaded in LMEM, just like in our application. After synthetizing it, we measured the times of streaming data/computing output in the DFE. Since there wasn't a real computation, the results we got were a good approximation of the effective streaming data time. By comparing these times with the previous ones, it turned out that, as we thought, the streaming of data was the real bottleneck. A possible solution was to move more arrays in LMEM but was not so easy since some of them were edited at each iteration of the loops in prop_source and in migrate_shot, as stated in part 2. However, another important fact rose during this analysis: we comparated the execution times of the sub-routine do_step in both original RTM application (without pragma) and in our implementation. The result was that the time required by the original RTM application for each do_step execution is almost equals to the time required by our implementation to prepare the arrays px/py, so even though we had solved the problem of the data streaming, our application would still have been slower that the original one.

5 Final Notes & Conclusions

In all the the approaches and tests we have done, we have always used the *rtm_parameters_small.txt* data starting from a "clean" situation (no RECEIVER_FILE). Once the last approach was completed, we made several tests on the original RTM application without pragma, with pragma and our implementation to see how the execution times scaled according to the data dimensions. Hence, we run all these three different versions of the RTM application with the three provided datasets. Here the results:

	RTM w/o pragma	RTM with pragma	RTM on DFE	#calls (do_step/do_step_damping)
small data	181,5s	49,5s	620,25s	4000
medium data	6232,5s	1747,5s	9698,25s	10000
large data	19926s	10756,5s	93102,75s	12000

Table 3: Times table (no RECEIVER_FILE)

Here a graphic representation of the previous data:

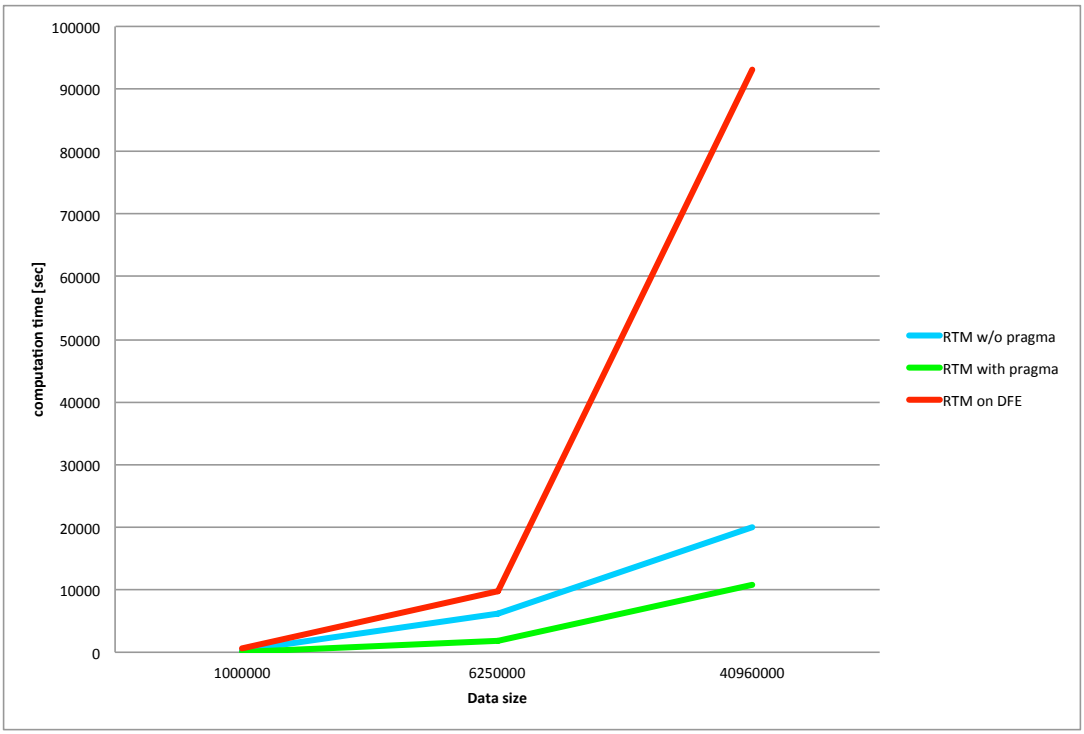

Figure 6: 2D graphic (x: data size, y: time in seconds)

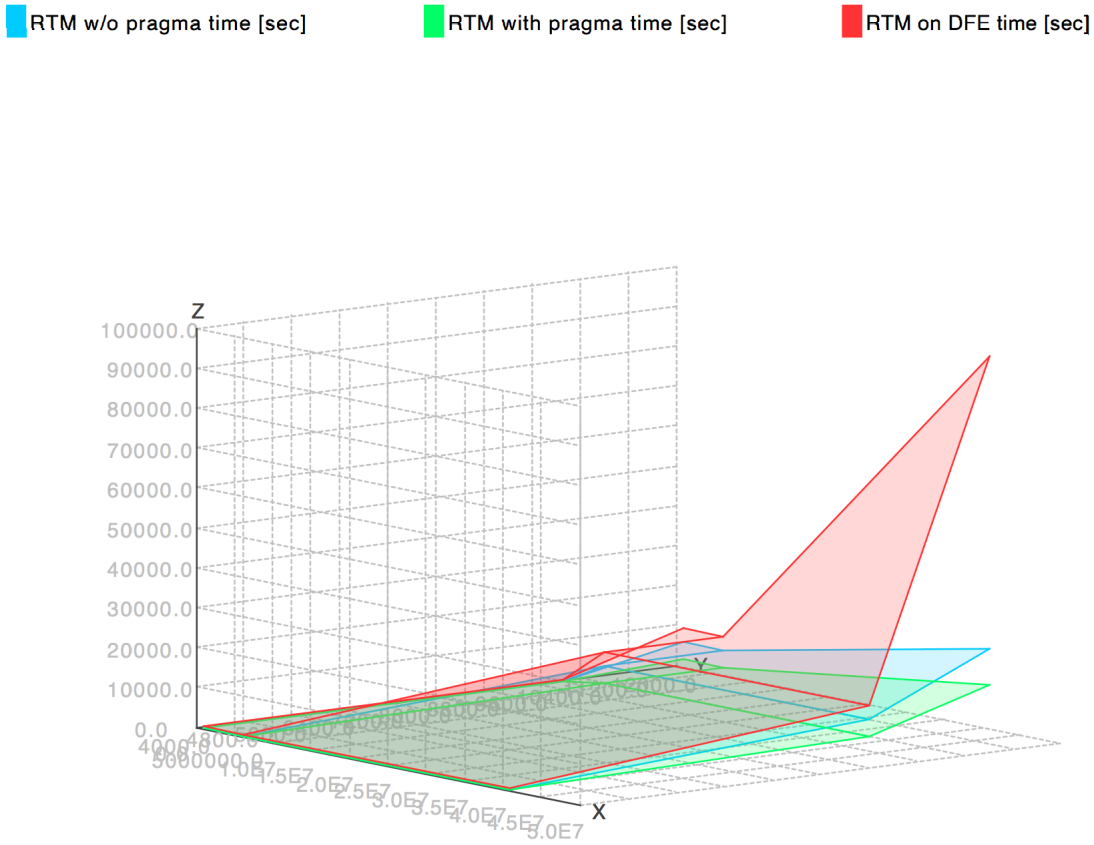

Figure 7: 3D graphic (x: data size, y: #calls, z: time in seconds)

As we can see, our approach is the slowest of the three variants. Hence, the idea of this project of translating in hardware just few sub-routines of RTM application is feasible but unsatisfying; indeed, to improve the performance, it would be better to change almost all the RTM code to make it more suitable for an hardware implementation. More in details, it would be necessary to move in hardware not only do_step and do_step_damping sub-routines, but also their callers (prop_source and migrate_shot), at least the loop inside them (the heaviest computational part) and, of course their control constructs (if/else, switch, loop conditions). In this way, it would be possible to keep the data "near to" the FPGA (inside the LMEM) to avoid a continouos streaming a data from CPU to FPGA via PCI Express.