HPPS Project: RTM – Maxeler

Enrico Deiana & Emanuele Del Sozzo

RTM – What it is

- 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.

RTM – How it works: main

```
main(){
        create_params_fields_data();
        if(!RECEIVER_FILE){
                prop_source();
                CREATION_RECEIVER_FILE();
        }
        prop_source();
        migrate_shot();
        dump_image_to_file();
        clean();
}
```

RTM – How it works: prop_source

```
prop_source(){
	for(i=0; i<NT; i++){
		add_source(SOURCE_CONTAINER);
		if(!RECEIVER_FILE){
			do_step_damping(P, PP, SOURCE_CONTAINER);
			extract_data();
		}else{
			do_step(P, PP, SOURCE_CONTAINER);
		}
		FLIP_P_PP();
	}
}
```

Small data: NT=1000
Medium data: NT=2500
Large data: NT=3000

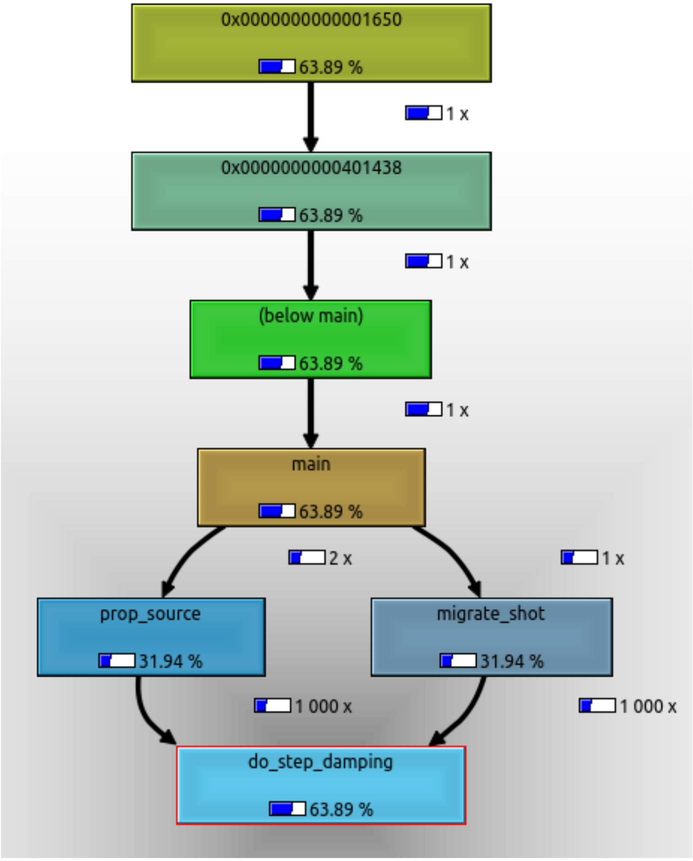
RTM – How it works: migrate_shot

```
migrate_shot(){
	for(i=0; i<NT; i++){
		clean_source();
		do_step(P, PP, SOURCE_CONTAINER);
		add_data();
		do_step_damping(P, PP, SOURCE_CONTAINER);
		image_it();
		FLIP_P_PP();
	}
}
```

Small data: NT=1000
Medium data: NT=2500
Large data: NT=3000

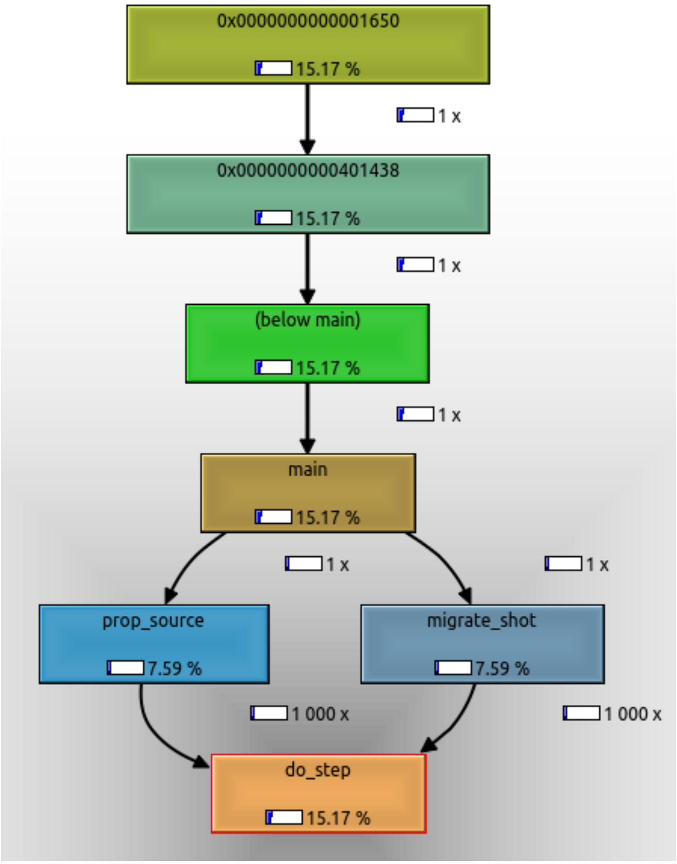
Profiling: callgrind

do_step_damping:



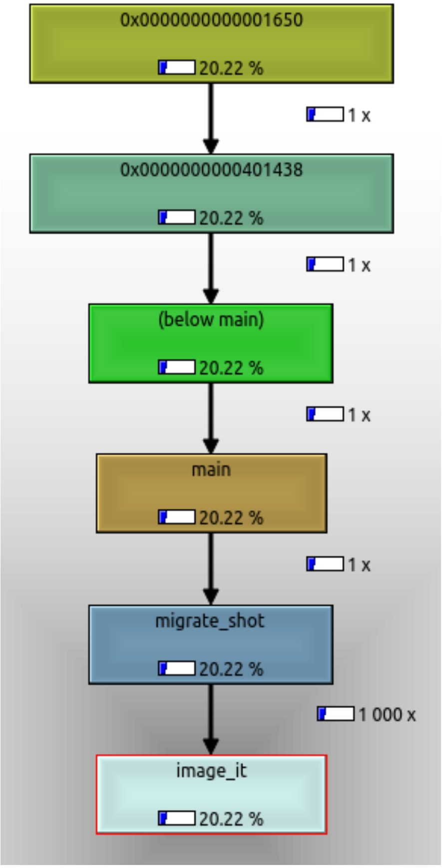
Profiling: callgrind

do_step:



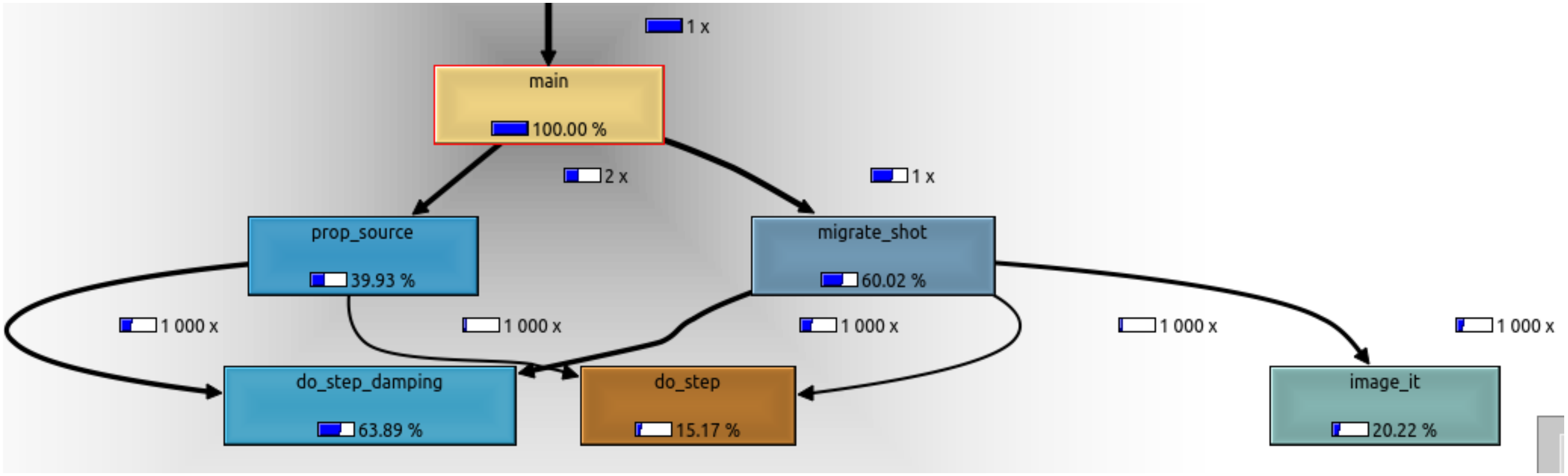
Profiling: callgrind

Image_it:



Profiling: callgrind

main:



Profiling: gprof

- 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%).

Approaches: do_step code

```
void do_step(float *__restrict p, float *__restrict pp, float *__restrict dvv, float *__restrict
    source_container){
    int i3;  //Indexes
    int n12=n1*n2;

#pragma omp parallel for
    for(i3=ORDER; i3 < n3-ORDER; i3++){   //Loop over slowest axis
        int i1;
        int i2;
        for(i2=ORDER; i2 < n2-ORDER; i2++){ //Loop over middle axis
            for(i1=ORDER; i1< n1-ORDER; i1++){  //Loop over fast axis
                //Wavefield update
                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]*
                    (
                        p[i1+i2*n1+i3*n12]*c_0
                        +c_1[0]*(p[(i1+1)+(i2  )*n1+(i3  )*n12]+p[(i1-1)+(i2  )*n1+(i3  )*n12])
                        +c_1[1]*(p[(i1+2)+(i2  )*n1+(i3  )*n12]+p[(i1-2)+(i2  )*n1+(i3  )*n12])
                        +c_1[2]*(p[(i1+3)+(i2  )*n1+(i3  )*n12]+p[(i1-3)+(i2  )*n1+(i3  )*n12])
                        +c_1[3]*(p[(i1+4)+(i2  )*n1+(i3  )*n12]+p[(i1-4)+(i2  )*n1+(i3  )*n12])
                        +c_1[4]*(p[(i1+5)+(i2  )*n1+(i3  )*n12]+p[(i1-5)+(i2  )*n1+(i3  )*n12])
                        +c_2[0]*(p[(i1  )+(i2+1)*n1+(i3  )*n12]+p[(i1  )+(i2-1)*n1+(i3  )*n12])
                        +c_2[1]*(p[(i1  )+(i2+2)*n1+(i3  )*n12]+p[(i1  )+(i2-2)*n1+(i3  )*n12])
                        +c_2[2]*(p[(i1  )+(i2+3)*n1+(i3  )*n12]+p[(i1  )+(i2-3)*n1+(i3  )*n12])
                        +c_2[3]*(p[(i1  )+(i2+4)*n1+(i3  )*n12]+p[(i1  )+(i2-4)*n1+(i3  )*n12])
                        +c_2[4]*(p[(i1  )+(i2+5)*n1+(i3  )*n12]+p[(i1  )+(i2-5)*n1+(i3  )*n12])
                        +c_3[0]*(p[(i1  )+(i2  )*n1+(i3+1)*n12]+p[(i1  )+(i2  )*n1+(i3-1)*n12])
                        +c_3[1]*(p[(i1  )+(i2  )*n1+(i3+2)*n12]+p[(i1  )+(i2  )*n1+(i3-2)*n12])
                        +c_3[2]*(p[(i1  )+(i2  )*n1+(i3+3)*n12]+p[(i1  )+(i2  )*n1+(i3-3)*n12])
                        +c_3[3]*(p[(i1  )+(i2  )*n1+(i3+4)*n12]+p[(i1  )+(i2  )*n1+(i3-4)*n12])
                        +c_3[4]*(p[(i1  )+(i2  )*n1+(i3+5)*n12]+p[(i1  )+(i2  )*n1+(i3-5)*n12])
                        ))+source_container[i1+i2*n1+i3*n12];
            }
        }
    }
    return;
}
```

Approaches: first simple approaches

- For each 3D cross, copy it into an array,
- Send it to the DFE,
- Return the value of pp.

- Feasible but useless (not scalable, not exploiting the machine structure)

Approaches: LMEM Blocked 3D

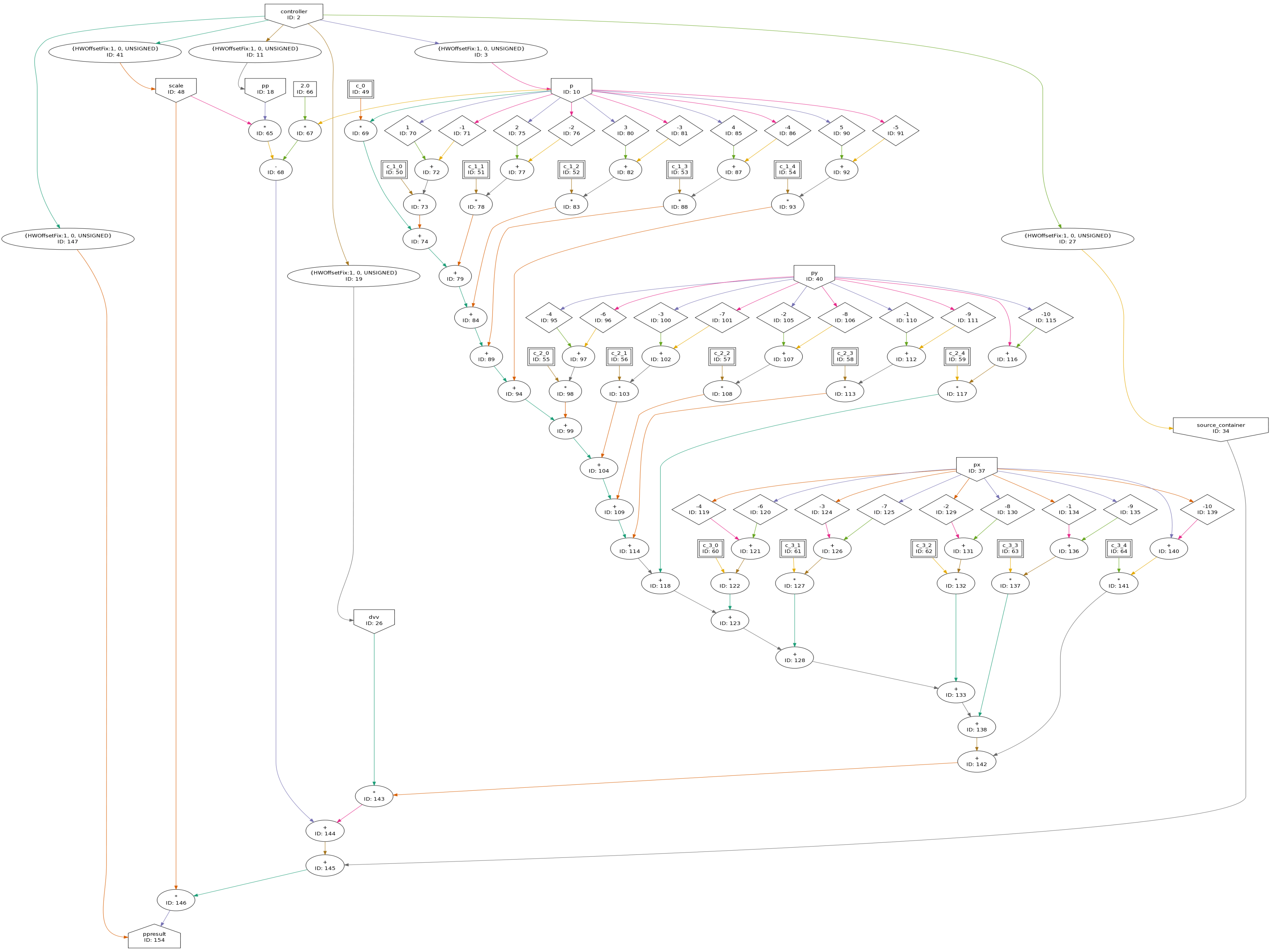
- Move the array p into LMEM as a 3D block,
- Select its subcubes of stencil size and compute pp values.
- Issues:
  one of the sizes of p must be multiple of 384 bytes,
  one of the sizes of subcube must be multiple of 384 bytes,
  one of the offsets must be multiple of 384 bytes.

Approaches: Stalling Streams

- Linearize p into three arrays according to the three dimensions x, y, z (in fact, px and py contain the linearization of the 3D cross that moves along z).
- Sinchronize the flowing of the streams to compute pp values.
- Two versions: not-optimized, optimized.

Approaches: Stalling Streams

- Not-optimized:
  only do_step moved in hardware,
  loading of DFE at each call of do_step.
- Optimized:
  do_step and do_step_damping moved in hardware,
  DFE loaded just once,
  px, py, controller, scale created once,
  controller never changes, hence load into LMEM.

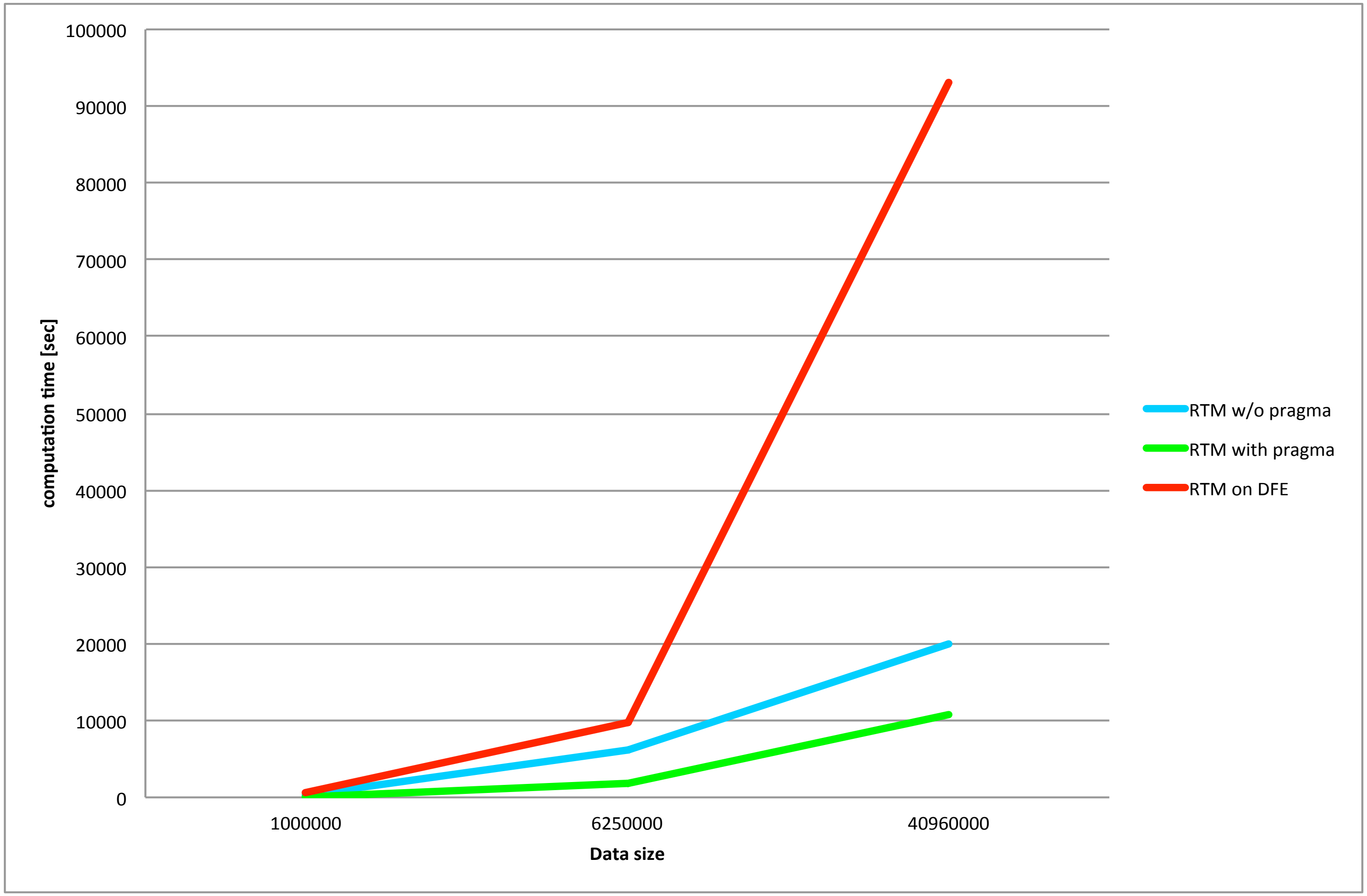


controller
ID: 2
{HWOffsetFix:1, 0, UNSIGNED}
ID: 41
{HWOffsetFix:1, 0, UNSIGNED}
ID: 11
{HWOffsetFix:1, 0, UNSIGNED}
ID: 3
scale
ID: 48
pp
ID: 18
2.0
ID: 66
c_0
ID: 49
p
ID: 10
{HWOffsetFix:1, 0, UNSIGNED}
ID: 147
{HWOffsetFix:1, 0, UNSIGNED}
ID: 19
{HWOffsetFix:1, 0, UNSIGNED}
ID: 27
py
ID: 40
source_container
ID: 34
px
ID: 37
dvv
ID: 26
pperesult
ID: 154

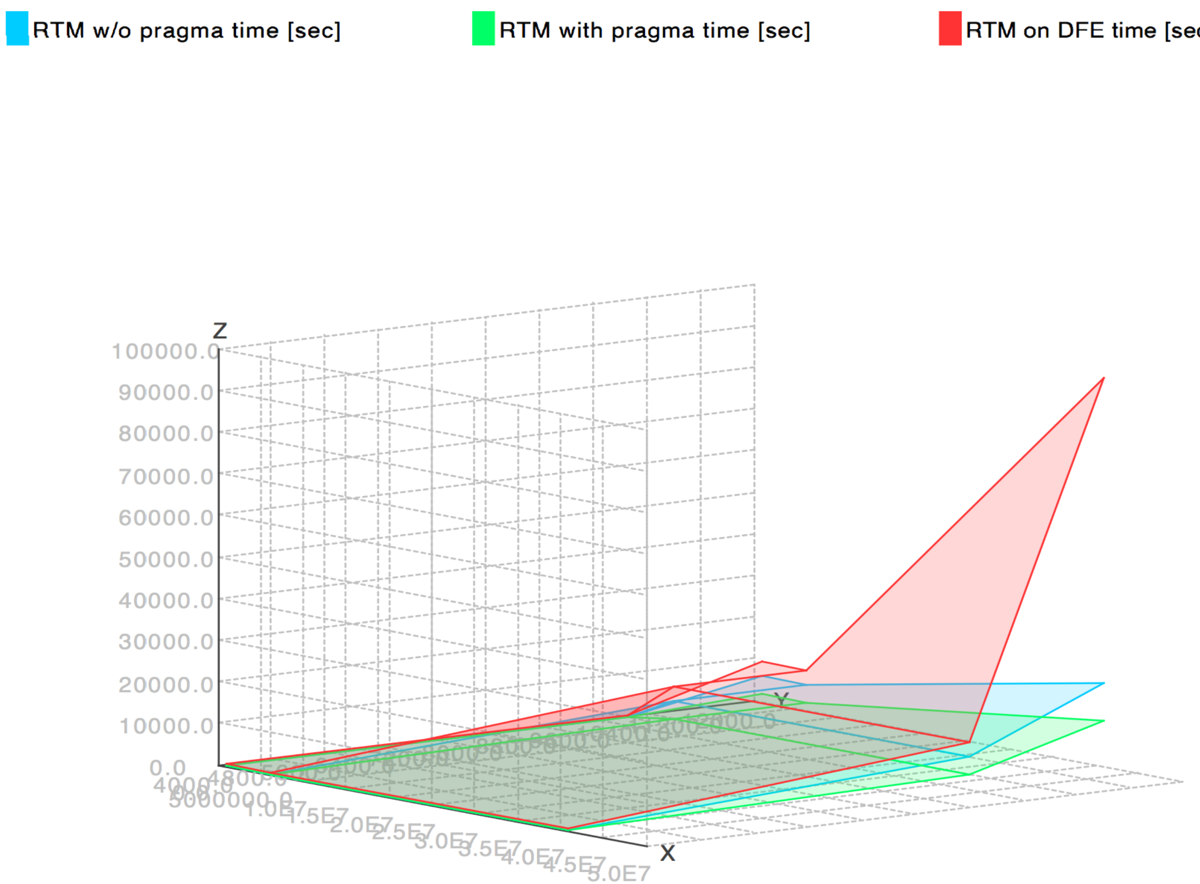
RTM applications comparison:

	RTM w/o pragma	RTM with pragma	RTM on DFE	#calls
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

RTM applications comparison:



RTM applications comparison:



x: data size
y: #calls
z: time in sec

Conclusions

- Our approach is the slowest one.
- The idea of this project of translating in hardware just few sub-routines of RTM application is feasible but unsatisfying.
- To improve the performance, it would be better to change almost all the RTM code to make it more suitable for an hardware implementation.
  It would be necessary to move in hardware not only do_step and do_step_damping sub-routines, but also prop_source and migrate_shot (and all their contents).
  In this way, it would be possible to keep the data "near to" the FPGA (inside the LMEM) to avoid a frequent streaming of data from CPU to FPGA via PCI Express.

Thanks for the attention!