Information and Communication Technologies (ICT) Programme
Project N^{o}: FP7-ICT-287804

D1.1: Requirements of FASTER Applications
WP1: Requirements and Evaluation Criteria

Author(s): Mathieu Robart (ST)
Oliver Pell (Maxeler)
Yannis Papaefstathiou (Synelixis)
Andreas Brokalakis (Synelixis)

Status -Version: Version 1.0 (Final)
Date: February 28, 2012
Distribution - Confidentiality: Public
Code: FASTER_D1_1_STM_FF-20120228

Abstract:
This document reviews the various requirements of the test applications provided by the three industrial beneficiaries. These applications cover different targeted application domains, from high-performance computing (HPC), rendering, to embedded computations and image processing.

Disclaimer

The FASTER Consortium is the following:

Beneficiary Number	**Beneficiary name**	**Beneficiary short name**	**Country**
1(coordinator)	Foundation for Research and Technology – Hellas	FOR	Greece
2	Chalmers University of Technology	CHT	Sweden
3	Imperial College London	IMP	UK
4	Politecnico di Milano	PDM	Italy
5	Ghent University	GNT	Belgium
6	Maxeler	MAX	U.K.
7	ST	STM	Italy
8	Synelixis	SYN	Greece

Document Revision History

Date	**Issue**	**Author/Editor/Contributor**	**Summary of main changes**
December 12, 2011	0.1	Mathieu Robart	Initial draft
January 30, 2012	0.2	Andreas Brokalakis	Synelixis NIDS application
January 31, 2012	0.3	Mathieu Robart	ST contribution integration
February 1, 2012	0.4	Mathieu Robart	Maxeler contribution integration
February 9, 2012	0.5	Mathieu Robart	Restructuration following Milan meeting
February 16, 2012	0.6	Mathieu Robart	Final draft
February 17, 2012	0.7	Mathieu Robart	
February 28, 2012	1.0	Mathieu Robart	Final version

Table of contents

1. Introduction

(CONTRIBUTORS: ALL)

This document reviews the domains of application covered by the FASTER project, along with the various requirements of test applications corresponding to these domains and provided by the three industrial beneficiaries. These applications cover different targeted application fields, from high-performance computing (HPC), workstation-class processing such as rendering, to embedded computations and image processing.

2. HPC (High Performance Computing) domain

2.1. Domain scope & requirements

HPC (High Performance Computing) addresses highly complex computational problems, due to the large amounts of data to process and/or the complexity of the involved calculation. Such tasks are typically performed on supercomputing-class systems or clusters, or on optimized parallel architectures when suitable.

Key characteristics of HPC problems are:

- Large scale - an HPC system will typically consist of many compute nodes (ranging from dozens to tens of thousands).
- Long runtimes - jobs will run for hours up to months at a time
- High computational requirements - in aggregate, arithmetic or other processing requirements are much higher than in desktop applications. However, these can be delivered either by a number of very powerful compute nodes or by a larger number of less powerful compute nodes.
- High memory requirements - many HPC problems utilize large volumes of data in memory, which can be located in each node or spread across many compute nodes. Within each node, the computations generally require very high bandwidth to memory.
- Limited user interactivity - due to the long running nature of most HPC applications, even on the largest supercomputers, user interaction is normally relatively limited, for example setting up a job and then awaiting its completion.
- Fast interconnect - HPC systems often have faster inter-node interconnects compared to standard computer servers or desktop systems. This is required to satisfy inter-node communication during compute jobs execution. At the same time many HPC jobs may be at least to some degree embarrassingly parallel and require minimal inter-node bandwidth.

Because of the large scale of HPC systems, some concerns that are only peripherally important in desktops become much more significant:

- Physical size - since there will be many compute nodes in an HPC system, each node should be as small as possible to minimize the size of the overall system. Space usage has an impact on build and running costs, both directly (e.g. rent) but also less directly (e.g. increased maintenance personnel required, longer/more complex cabling).
- Power consumption - large-scale HPC installations already consume many megawatts, and power consumption is a first-order design constraint. Even for smaller scale HPC systems (a few dozen nodes) there is typically a limit on the power available.
- Reliability and fault tolerance - while the failure of an individual node may be relatively rare, with many nodes the probability that one will fail is dramatically increased and with long execution times there is a risk that complete jobs may never run to completion without encountering an error. Nodes themselves must be as reliable as practical and the HPC system itself must tolerate failures, for example by supporting checkpointing and restarting of jobs.

2.2. Sample application: Reverse Time Migration (RTM)

(CONTRIBUTORS: MAXELER)

2.2.1. Application description

Some of the most computationally intensive geoscience algorithms involve simulating wave propagation through the earth. The objective is typically to create an image of the subsurface from acoustic measurements performed at the surface. To create an image of the subsurface, a low frequency acoustic source on the surface is activated and the reflected sound waves are recorded by (typically) tens of thousands of receivers. We term this process a “shot”, and it is repeated many thousands of times while the source and/or receivers are moved to illuminate different areas of the subsurface. The resulting dataset is dozens or hundreds of terabytes in size. The problem of transforming this dataset into an image is computationally intensive and can be solved with a variety of techniques. Reverse Time Migration (RTM) is a high end technique for generating images of the earth and is used in complex geologies to give detailed subsurface images.

The concept behind RTM is relatively simple. We start with a known earth model. This earth model might be simply acoustic velocity, but could also be anisotropic, elastic, or even visco-elastic. Scientists conduct two modeling experiments simultaneously through the earth model. Both attempt to simulate the seismic experiment conducted in the field—one from the source’s perspective and one from the receivers’ perspective. The source experiment involves injecting our estimated source wavelet into the earth and propagating it from *t=0* to our maximum recording time *tmax*, creating a 4D source field *s(x, y, z, t)*. Typical values for x,y,z,t are ~1000-10000. At the same time, we conduct the receiver experiment. We inject and propagate our recorded data starting from *tmax* to *t0*, creating a similar 4D volume *g(x, y, z, t)*. We have a reflection where the energy propagated from the source and receiver is located at the same position at the same time, thus an image can be obtained by summing the correlation of the source and receiver wavefield at every time point and every shot, i.e.:

$$I(x, y, z) = \sum_{j \in shots} \sum_{t=0..tmax} S_j(x, y, z, t) \times G_j(x, y, z, t)$$

We frequently wish to collect *subsurface offset gathers*, cross-correlating source and receiver wavefields by various shifts:

$$I(h, x, y, z) = \sum_{j \in shots} \sum_{t=0..tmax} S_j(x - h, y, z, t) \times G_j(x + h, y, z, t)$$

h is typically a few dozen, and it is this subsurface offset gather calculation which provides a compelling use-case for partial reconfiguration since it is a significantly different calculation compared to the wave propagation processing. This can be seen in the pseudo-code for the Maxeler RTM shown in Figure 1.

The first step of the computation is to simulate the propagation of the source wave to *tmax*, then the computation is reversed and the source and receiver wavefields are propagated together from *tmax* to *t0*. This method involves 50% more computational effort than a naive direct implementation but is necessary to avoid the storage of large 4D state fields (which have sizes of many terabytes). There are two major computation kernels: *propagate* and *image*, which are fundamentally different. With sufficiently low-overhead dynamic (partial) reconfiguration, RTM could benefit substantially from time-division multiplexing of the FPGA resources between the *propagate* and *image* computations.

```
migrate_shot(shot_id) {
        src_curr = zeros(nx,ny,nz);     src_prev = zeros(nx,ny,nz);
        rcv_curr = zeros(nx,ny,nz);     rcv_prev = zeros(nx,ny,nz);
        image = zeros(nx,ny,nz,nh);

        model = load_earthmodel(shot_id);

        for t = 0 .. tmax {
                add_stimulus(shot_id, t, src_curr);
                propagate(src_curr, src_prev, model);
        }

        swap(curr_src, prev_src); // reverse time direction

        for t = tmax .. 0 {
                propagate(src_curr, src_prev, model);

                add_receiver_data(shot_id, t, rcv_prev)
                propagate(rcv_curr, rcv_prev, model);

                if (i % image_step == 0) // typically every 5-10 steps
                        image(src_curr, rcv_curr, image);
        }
}
```

Figure 1 - Pseudo-code for RTM

2.2.2. Requirements

2.2.2.1 Computation Complexity

RTM is computationally very intensive - $O(nshots \times ntimesteps \times nx \times ny \times nz)$ - however it has a regular computational structure. Both the *image* and *propagate* kernels apply fundamentally the same calculation to every point in the 3D problem domain, though there are some additional calculations needed at the boundaries.

2.2.2.2 Type of Processing

The main arithmetic operations for RTM are multiplications and additions, however the computation also puts significant pressure on the memory system both in terms of capacity and bandwidth. For nx=ny=nz=1000 and 50 subsurface offsets, the total memory required is over

200GB, while this data must be quickly read in/written out of the chip. The performance of most RTM implementations is memory-bandwidth limited.

2.2.2.3 Control Flow

Control for the RTM application is essentially the loop structures detailed in Figure 1. In a FPGA accelerated implementation these control structures remain on the CPU as software loops. The FPGA executes the iteration loop over *nx*, *ny*, *nz* and manages the control associated with applying differing operations at different points in the domain. In hardware this consists of counters keeping track of position, and comparators evaluating that position against special case conditions (e.g. boundaries).

The control flow of the application depends entirely on input parameters and is independent of the results of the computation, so can be completely predicted in advance.

2.2.2.4 Data Flow

The *propagate* or *image* kernels can be implemented as a streaming datapath with a feed-forward pipeline. There is no feedback of values within a single timestep.

2.2.2.5 Parallelism

There are several potential methods to parallelize RTM:

- **Shots** - Each shot represents typically many hours of computation and can be computed independently from any other shot, with the final images summed.
- **Domain blocks** - a single 3D domain for a single shot can be split into multiple domain blocks which can be processed by multiple processing chips. Each domain block must communicate boundary regions with the adjacent blocks.
- **Spatial points** - multiple adjacent spatial output points can be computed in parallel. These points share almost all of their input data.
- **Pipelining** - a highly optimized RTM datapath typically consists of many levels of pipelining which can be efficiently employed because of the lack of feedback loops.

Ideally we opt to parallelize *shots* over multiple nodes in a cluster, *domain blocks* over multiple processing chips within a single compute node (e.g. 4 FPGAs in a Maxeler MaxNode) and *spatial points* within a chip.

2.2.2.6 Memory Constraints

RTM is a memory-intensive algorithm. Two key features determine the memory consumption.

The first is the physics model employed. For Maxeler's RTM we use an acoustic isotropic model of the earth, which is described by one earth model parameter volume (*velocity*) and requires us to store two pressure states for each wavefield. This totals 5 volumes.

The image volume containing the computed subsurface offset images, is 4D with a size determined by the number of subsurface offsets being collected.

In CPU implementations, the data volumes are stored using 4-byte single precision floating point. In FPGA accelerated implementations we can compress the model volumes to 2-bytes per point. Figure 2 shows the memory required for RTM.

nh	**Total memory required (CPU)**	**Total memory required (FPGA)**
1	24GB	12GB
10	60GB	30GB
25	120GB	60GB
50	220GB	110GB

Figure 2- Table showing memory required for RTM with *nx=ny=nz=1000* and varying numbers of subsurface offsets (*nh*)

This memory contains the state of the calculation and must be maintained while the shot is computed. This means that any reconfiguration of the chip during computation must preserve the memory contents.

2.2.2.7 Targeted Performances

RTM is a performance-sensitive HPC application, however there are no real-time constraints. It is advantageous to be able to compute a result as quickly as possible. In real use, a timeframe of approximately 2 weeks to compute a full RTM is typical using conventional processors. An accelerated implementation should be at least 10x faster to be considered as an attractive alternative. Individual processing timesteps should take of the order of hundreds of milliseconds to a few seconds. If the FPGA is to be reconfigured to execute the *image* calculation, it should therefore be able to do so twice every *image_step* timesteps without significant performance penalty (i.e. of the order of once every second).

2.2.2.8 Reconfigurability

The characteristics of individual RTM runs depend heavily on the configuration. Some parameters commonly change every timestep or few timesteps:

- Type of computation (imaging or propagation): Every few timesteps the *image* calculation is applied, which is a different calculation compared to the *propagate* calculation.
- Stimulus data injection: Some values may be written into the domain (for example an initial stimulus wavelet, or receiver data recorded during a survey). The data values and position those values should be written to could change.
- Receiver data readback: Values may be read out of the domain to the CPU during a timestep. The position that these values should be read from may change.

Parameters that could change for every shot:

- Domain size (nx, ny, nz) and number of timesteps (nt): This effects both memory use on the node ($O(nx{\times}ny{\times}nz)$) and computational cost per node ($O(nx{\times}ny{\times}nz{\times}nt)$).
- Number of subsurface offsets (nh): This effects memory required to hold the offset image volumes and the amount of computation required to compute them.

Other parameters which change less frequently are:

- Physics model: this fundamentally changes the nature of the computation and the amount of storage.
- Size of convolution stencil operator: this can offer more or less accuracy at varying computational costs.

Parameters that change rarely (e.g. physics model) are probably most suited to full reconfiguration since there is little impact of doing so. Parameters that change more frequently are potential candidates for the use of reconfiguration. For example, micro-reconfiguration could be used to specialize the domain size into the accelerator configuration, or partial region-based reconfiguration could be used to switch between *image* and *propagate* calculations.

3. Desktop-class domain

3.1. Domain scope & requirements

This domain covers applications potentially running on desktop workstations. This covers general-purpose single processor systems (potentially composed of several cores), where performance requirements can be achieved through high performance CPUs, SIMD based on vector processing (e.g. Intel SSE2/3/4, ARM NEON) and multithreading with processes running in parallel on several cores. Parallelization can be also exploited using a GPU commonly present in such systems, through a dedicated API such as OpenCL or CUDA.

Key characteristics of HPC problems are:

- Single processor – a desktop workstation system will typically include a single processor, composed of several high frequency cores (2 to 10), each one able to run one or two threads in parallel.
- Average runtimes – a typical execution time runs between real-time and few hours (lower than HPC, much higher than embedded).
- High computational requirements – some types of computation (arithmetic) can be compared to HPC ones, however on a lesser scale. Typical optimizations include use of SIMD extensions natively supported by the CPU, multithreading over the several cores and computation off-loading over the available GPU using a compute API. L2 cache supported by the CPU can also greatly improve the efficiency of code.
- Memory size – a desktop workstation system typically supports several GBs of RAM with different patterns of access. When suitable, dedicated graphics memory with high bandwidth can be also used by the associated GPU.
- User interface – most applications make use of some form of interactivity, through the use of interfaces or visual feedbacks.

Potential concerns:

- Power consumption – one of the main restrictions to the development of faster CPUs and GPUs is power consumption and heat dissipation. Power consumption can also be major concern regarding mobile systems (however, less important than for embedded devices).
- Cost – depending on the area of application (e.g. mainstream applications), the cost of ownership can also be a concern, as desktop systems are seen today as commodity.

3.2. Sample application: Ray Tracing

(CONTRIBUTORS: ST)

3.2.1. Application description

In future graphics applications (games, visualization, etc.), it will be important to achieve photorealistic rendering in a coherent manner, in order to greatly improve picture quality with an ever-increasing scene complexity, with support for real reflection, soft shadows, area light source, indirect illumination, etc. This is a computationally intensive problem, addressed by the increasing interest in real-time global illumination approaches. Within FASTER, STM is developing an OpenCL global illumination scheme. This system should be flexible enough to help accelerating different algorithms based on ray casting (ray tracing, path tracing, Monte Carlo ray tracing, etc.).
A 3D scene is described mathematically, using simple primitives such as triangles, polygons, spheres, cylinders, and more. The properties of each primitive, such as position, orientation, scale

and optical properties, are described by the scene. A virtual camera is placed into the scene, and an image is rendered accordingly in casting rays simulating the reverse path of ray of lights, from the origin of the camera through every pixel of its virtual focal plane. The color of a pixel is determined by the potential intersections of the primary ray casted through it, with the 3D scene. Photorealistic realism can be achieved when sufficient rays are casted, simulating with fidelity the behavior of the light. Simulating the proper materials behaviors is also paramount.

3.2.2. Requirements

3.2.2.1 Computation Complexity

The raytracer application is a fairly flexible application, with a complexity that can be tailored to the quality of the image to render. Three computationally intensive stages can be identified:

- *Acceleration structure setup*: all the geometric primitives can be potentially seen through every pixel of the rendered image; as a consequence, all need to be tested for intersection. This approach can be acceptable for simple scenes with very low geometry complexity (~10s of primitives), but become impracticable for more complex databases with a complexity of $O(n^2)$. As a consequence, an acceleration data structure should be built when the scene to render is loaded, e.g. a voxel grid. With such approach, each primitive must be referenced by at least one voxel of the grid, in comparing its axis-aligned bounding box with the location of the voxels of the grid. The cost of this setup will be greatly compensated during the rendering phase, when only a potentially low subset of primitives will have to be intersected by each ray casted through every pixels of the rendered image. The cost of the structure traversal must also be taken into account.
- *Ray-primitive intersection*: the rendering of an image demands the computation of the intersection of geometric primitives of the scene (polygons, spheres, cylinders, etc.) with rays casted through each pixel. Significant amount of the computation will be dedicated to this sole purpose. The complexity of the intersection computation depends itself on the nature of the primitive: the intersection ray-sphere is cheaper to compute than ray-polygon for example. As a consequence, the corresponding amount of computation complexity is very scene-dependent.
- *Shading computation*: when a visible intersection is found, its shading must be evaluated. Depending on the material of the intersected primitive, this stage can be trivial or extremely complex. A flat material ignoring shadowing will just return a color modulated by a dot product between the primitive normal and the incident ray. A glossy material taking account shadowing will potentially generated many additional rays (shadow rays, secondary rays) that will be casted again against the 3D scene. The results of these intersections will be used to define the final color for that primary ray.

The number of primary rays to cast is itself dependent of the resolution of the image to render. For example, a VGA image (640x480 pixels) without anti-aliasing will demand 640x480 = 307200 primary rays. A full HD images with anti-aliasing (e.g. 16 samples per pixels) will demand 1920x1080x16 = 33177600 rays (or x108 more primary rays).

For accuracy reason, all the computations are performed in floating point. 32-bit single precision is acceptable, but 64-bit floating points could be useful for scenes with important difference in scales, at the cost of performances on some architectures.

3.2.2.2 Type of Processing

A lot of vector computations (normalization, dot product, scaling, etc.) can take place, based on vectors, 3D points or normals parameters. Intersection computations are also mathematically intensive, with potential use for square roots. Trigonometric functions are also extensively used for computing random directions or rotation transformation, for example.

3.2.2.3 Control Flow

A lot of tests are performed in order to generate the rendered pixels and to cut short the redundant computations. This can introduce a disbalance in the cost of consecutive pixels, when one hits a complex surface while the next misses it.

The rendering is performed in looping across all pixels of the screen and casting rays. For each ray, the process is looping across all considered primitives in order to find the closest intersection.

3.2.2.4 Data Flow

A 3D scene uses a voxel grid. A voxel grid contains a 3D array of cells (or voxels), each one containing an array of pointers to the primitives contained in that cell. The primitives themselves contain the geometry information permitting to compute their intersection with a ray (location, orientation, size, etc.). Each primitive also points to a shared material describing its optical properties (reflectance, color, emissivity, etc.).

A 3D scene maintains also a list of primitives used as potential light sources (i.e. primitives with an emissive material).

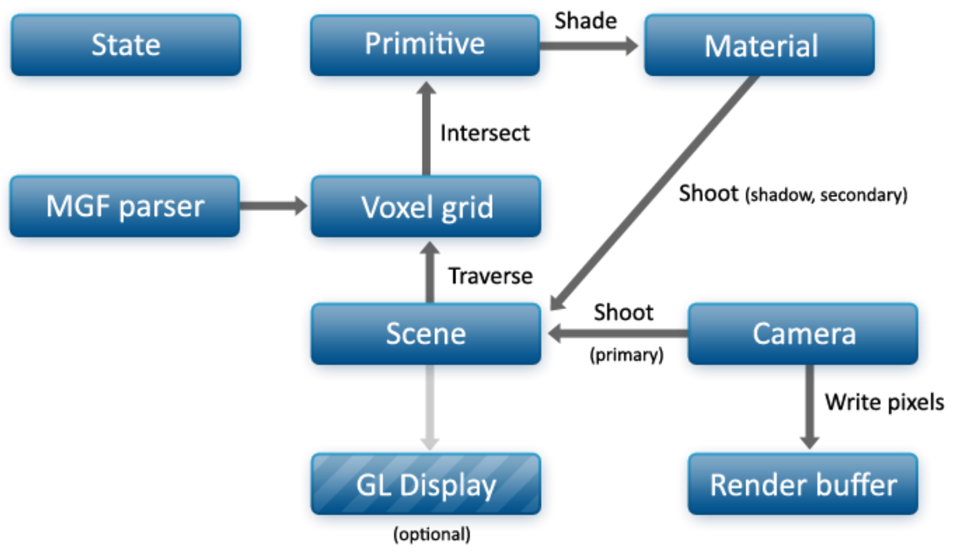

Figure 3 – raytracer data flow

The camera uses a render buffer where the final image will be generated. During rendering, the camera creates a single primary ray originating at the camera location, and pointing in the direction of a single sample inside a given pixel of a virtual focal plane. The 3D scene is interrogated for potential intersection with this primary ray.

The 3D scene will interrogate the voxel grid for intersection. The first voxel encountered by the primary ray is identified, and the intersections between the primary ray and all the primitives contained by the voxel are computed. If no valid intersection is found, the process is repeated with the next voxel along the ray path until reaching the limit of the voxel grid, or until an intersection is found.

If an intersection with a primitive is found (the closest of all the intersections with the primitives of a single voxel), its shading must be computed using the material used for the corresponding primitive. This shading computation may generated additional rays (e.g. shadow rays), casted in turn into the scene following a similar process. The result of the shading stage is a color associated with the intersection point. For a primary ray, this color is used to compute the color of the original pixel by the camera. For a secondary ray (or shadow ray), this color is used to compute the color of the surface hit by the primary ray.

3.2.2.5 Parallelism

Raytracing is an *embarrassingly* parallel problem. Each primary ray (i.e. each pixel sample) can be processed independently and as consequence in parallel. By extension, each path (sequence of rays from a pixel) can be also processed in parallel.

The scene description, and its associated voxel grid, must be shared as all primitives are potentially seen from every pixel.

Random parameters can be generated during rendering (e.g. for jittering the location of the samples inside a pixel in order to reduce aliasing, or to generated a sample inside a primitive). However, if parallel processing is used, repeatability may become an issue if care is not taken. A possible solution is to generate pre-computed random grids that will be used during rendering as shared assets between processes.

The full version of the application (D5.4) will have some computation steps ported to OpenCL in order to exploit this parallelism in an explicit way, relying on CL kernels for heterogeneous platform support.

3.2.2.6 Memory Constraints

The memory used by the 3D scene is obviously dependent on its complexity: the more primitives are defined, the more memory is required.

A render buffer is used to store the computed pixels. The size of this buffer is directly linked to its resolution and must store RGB(A) values for each pixel. For example, 8-bit VGA image will require 640x480x3 bytes = 900Kb, whereas a full HD fp image will require a buffer of 1920x1080x3x4 bytes = 23.7Mb.

The voxel grid itself stores pointers to primitives. Its resolution can be parameterized: a lower resolution will demand less memory, providing a better traversal time but possibly more intersection computations and worst overall performances. The balance between the size of the voxel grid and the complexity of the 3D scene may be critical, and different per scene.

No texture is used in the current version of the raytracer.

3.2.2.7 Dependencies

An optional module based on OpenGL is used for visualization of the scene, position of the camera, etc. However, this functionality is not required for the computation of the rendered image itself and can be removed via compilation options or code #define. Without this support, the application is self-dependent.

The full version of the application (D5.4) will have some computation steps ported to OpenCL.

3.2.2.8 Targeted Performances

The image rendered by raytracing should be produced as fast as possible. Real time may be a target for scenes with low complexity, or low resolution. However, such an ideal target may not be reached for more complex setup. Consequently, some form of hardware acceleration can be necessary to obtain a satisfying frame rate for scene with average complexity.

3.2.2.9 Reconfigurability

The raytracing application is highly reconfigurable, depending of the input that is processed to the way the rendering is performed.

The workload is first dictated by the resolution of the rendered image. As at least one primary ray is casted per pixel (and potentially more for antialiasing, even reaching thousands in path tracing applications), the image size dictate the rendering complexity and the possible parallelization width. Image resolutions can vary from VGA (640x480) up to full HD (1920x1080) or more.

Antialiasing is performed in casting more than one ray per pixel. As stated in the previous point, this parameter has a direct influence on the workload of the application. Reasonable values can range between 4 and 16 samples per pixels.

A ray tracing application spends a lot of time in computing intersections between a ray and a geometric primitive. The nature of this primitive dictates the complexity of the intersection computation. A highly optimized solution will target a single type of primitive (e.g. triangles), with a possible hardware-accelerated support. However, the need for more complex surfaces (e.g. cone, sphere, polygons, or even high-order surfaces) demands a reconfiguration of the application in order to take into account these new primitives. High-order surfaces may even demand additional stages, e.g. tessellation.

Intersections are expensive to compute, and their number should be reduced to the minimum. For that effect, acceleration structures must be used in order to describe the scene in a more efficient manner. Voxel grids or kd-trees are possible solutions. The choice of the nature of this structure may depend on the type of scene to render, and the possible hardware support. For example, the traversal of a voxel grid could be accelerated by a dedicated device. The resolution of the voxel grid itself could influence some form of reconfiguration, according to the scene complexity for example. Additionally, a static scene needs a single construction of the acceleration structure when the scene is first described, and could be done as a pre-process. However, a dynamic scene with transformable objects (animated position, size, etc.) needs a constant update of the corresponding acceleration structure.

The color of a pixel is determined in following the path of a ray in the virtual 3D scene, after subsequent interactions with the intersected primitives. The depth of this path can be controlled by the user as a parameter to the application (in limiting it to 3 or 4 bounces for example), or computed analytically. Consequently, this choice has a direct consequence in the computation complexity of the rendering.

The shading of the point of a primitive hit by a ray must take into account the direct illumination that it receives. This evaluation is done in casting *shadow rays* in the direction of the light sources. If one shadow ray intersects something before reaching the targeted light source, an occlusion is detected and the rendered point is potentially in shadow. In general, light sources have an area, and consequently several shadow rays must be cast to sample it correctly. Once again, the choice of the number of shadow ray can be influenced by some application parameters, with a direct consequence on the quality of the rendering and its computation time (from a single shadow ray to hundreds or more per intersection).

In addition to the obstruction factor, the shading of an intersection demands the computation of a reflectance according to the material associated to the primitive. This computation can be very simple but unrealistic (based on simple dot products), or can involve complex shading models, based on physical properties of real materials, themselves relying on additional ray casting for evaluation of reflection or diffusion, for example. The choice of the shading model depends on the scene to render, on the capabilities of the application and on desired performances.

The size of the scene itself impacts the rendering time. Indeed, the more primitives compose the scene, the more intersections need to be computed. A correct acceleration structure tends to reduce the impact of the scene complexity, but the number of primitives has nevertheless an important influence on the rendering time. Typically, the number of primitives can range from 2 (e.g. test scene) to several millions.

The final rendered image has generally a high-dynamic range, encoded in floating point, in order to capture the full spectrum of the light intensity, from very bright light sources to dark penumbras. However, the result needs generally to be displayed on a screen with a limited gamut, and some post-processing is necessary (e.g. tone mapping). The presence of this last stage, and the quality produced, can be freely parameterized by the application itself, and modified subsequently in function of the output device targeted.

Finally, the level of parallelism that can be exploited also depends on the platform of execution. On a multi-core platform, the image can be decomposed in independent sub-regions, and each region is rendered independently by a single core. The scene data and the associated acceleration structure have to be shared between processes. On more dedicated platforms (or GPUs for example), each pixel can be processed independently by its own thread, reaching a very high level of parallelism.

4. Embedded domain

4.1. Domain scope & requirements

The term *embedded system* is used frequently nowadays to describe a wide variety of modern devices/applications. As a victim of its generality, there is not a precise definition available but a rather loose or subjective understanding of what actually can be characterized as an embedded system. A broad definition could be as simple as that[1]: *embedded computer system is any device that includes a programmable computer but is not itself intended to be a general-purpose computer*, or a little more detailed[2]: *an embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based.*

Following the above definitions, typical embedded systems include common domestic appliances (from clocks to fridges to televisions), portable devices, industrial control systems, cars, defense systems, telecommunications equipment and so on. Compared to more familiar computing systems, such as the personal computers (PCs), embedded systems are special purpose devices and are usually designed with certain requirements or restrictions that may not apply for other systems. For example, a medical device or an industrial control application may impose higher safety requirements than a typical electronic device.

Key characteristics of embedded systems are:

- Small scale systems – an embedded system is typically restricted both in resources and physical dimensions (there are however notable exceptions, e.g. in military applications)
- Short runtimes to real-time systems – embedded applications typically involve real-time systems or systems that have to respond within tight time frames
- Varied computation requirements – since embedded systems span from simple microcontroller-based electromechanical systems to high-performance highly complex ones, computational requirements vary by a large degree
- Low memory requirements – embedded systems are not meant to be general-purpose systems and therefore are designed with minimum memory requirements (this is also typically associated to cost reduction and physical area minimization)
- Fast interconnects – embedded systems typically employ fast and proprietary interconnects since they are closed systems that do not have to adhere to general-purpose interfaces or standards, as a way to minimize latencies and complexity.

Since embedded systems are special purpose devices, apart from the programmable hardware they include (in the form of a generic microcontroller, microprocessor or DSP), they often use custom hardware, mainly for two reasons. The first one is related to the fact that they have to operate under strict restrictions and requirements which simply cannot be met by software. For example, in a cell phone, it is not easy to implement all the communication protocol complexity in software since it has to operate under certain space, energy and thermal constraints. Thus dedicated circuits are used to offload these computations.

The second reason for using dedicated hardware is attributed to the fact that the designer of an embedded system has almost complete knowledge on how the device will be operated and

[1] Wayne Wolf, “Computers as Components: Principles of Embedded Computing System Design”, 2nd Edition, Morgan Kauffman Publications, 2008.

[2] Michael J Pont, “Patterns for Time-Triggered Embedded Systems”, TTE Systems, ACM Press Books, 2011.

furthermore the functionality of the device may be fixed. Since dedicated hardware can be very efficient in terms of performance and energy consumption, the aforementioned allow for systems that can be made cheaper and better performing. Continuing the previous example, the designer knows the communication protocol that will be used by the cell phone, as well as all the basic functionality that the phone has to offer and thus by using dedicated circuits he can meet energy and performance requirements easier and cheaper (e.g. he can use simpler components and smaller batteries).

From the above, one has to focus on the following key aspects of embedded systems: they are closed systems (usually not meant to be user expandable or serviceable) and they are tightly designed to meet certain specifications with cost being a major concern. That means that during design, an embedded system can be tailor made to a specific structure suitable for a particular application. Thus components can be directly interconnected, since they do not have to adhere to general-purpose interfaces and topologies, minimizing latencies and complexity. On the other hand, as the application domain is strictly characterized, memory and other resources are kept to the bare minimum required by the application. Therefore embedded systems typically have access to fewer resources than their general-purpose desktop-class counterparts.

The requirements of embedded systems can be summarized in the following list:

- Operation under strict predefined specifications
    - Real-time systems
    - Reliable systems
    - Specific physical area, energy consumption and computational performance requirements that may precondition the use of high-performance multi-core processors and hardware accelerators (e.g. ASICs, FPGAs, GPUs)
- Minimum component count
    - Low memory requirements (small memory sizes)
    - Reduced IC count and manufacturing complexity
- Highly cost sensitive
    - Component costs
    - Energy consumption costs
    - Design reuse, product upgradability

The major drawback of using dedicated hardware and interconnects is the lack of flexibility either during design or after the system has been deployed. The former restricts the production of rather generic systems that can be differentiated or used in other applications simply by software changes. The latter is also important since the functionality of the device may have to be altered after it has been delivered.

From the above, it is understandable that for certain applications where there are strict performance and energy constraints that cannot be met by commodity or special purpose programmable processors, flexibility has to be sacrificed through the use of dedicated hardware. A solution that has been successfully presented for this problem is the use of configurable logic, primarily through the use of FPGA devices. FPGAs can provide performance (as well as energy consumption) close to the one offered by dedicated ASIC circuits, while they can be reconfigured after they have been deployed, thus they form a middle ground between ASICs and software, offering the best of both.

An application field that has embraced the use of FPGAs is the network embedded systems one. Typically, network embedded systems require high levels of performance and commonly use dedicated hardware along with high-performance processor cores. The need to support a multitude (as well as evolving) communication protocols, to rapidly respond to critical changes and to deploy flexible systems so as to protect the investments of customers are among the key drivers of adopting FPGA technology in those systems. In order to keep up for example with high data bit rate communications while supporting encrypted messages, modern routers and other network devices employ hardware implementations of the most popular cipher algorithms. As, however, those are susceptible to vulnerabilities, in order to keep those systems safe, changes have to be made as soon as those vulnerabilities are discovered and this is only feasible through the use of reconfigurable devices.

The aforementioned reveal a possible weakness of using FPGA devices. More specifically, the multitude of functions that may have to be implemented in an FPGA can be overwhelming thus requiring either multiple devices or the use of the most expensive larger devices. On the other hand, while small changes may be required, a device has to be reconfigured in order to apply those changes. This results in (potentially) significant down-time and requires large storage resources for configuration bitstreams. To alleviate these problems, modern FPGAs have started offering partial reconfiguration options, which can be applied very effectively in those scenarios.

A certain class of network embedded systems that can deeply benefit from the use of FPGA devices is Network Intrusion Detection Systems (NIDS). NIDS systems are becoming essential in modern networks as security is escalating to be one of the most important aspects. They typically operate under very high performance requirements, thus requiring the use of dedicated hardware, however they have to be constantly updated to detect newly discovered vulnerabilities and attack methods. That is the reason we consider it a prime candidate for FASTER. The application and its requirements will be presented in Section 4.2.

A second application field that rapidly adopts FPGA technology, is the Image Analysis one. Machine vision systems are becoming widespread and they are used in multiple domains, such as surveillance, navigation, medical and control systems. A common requirement in these applications is the need to analyze in real-time (or almost real-time) large image data structures, thus the performance requirements are significant. Graphics Processing Units (GPUs) have often been used as accelerators for these systems, since they offer high performance and they are programmable, therefore they can adapt to different system requirements or incremental changes/updates of a certain system. However the use of GPUs presents certain restrictions. First, they are costly both in terms of actual component cost and in terms of increased second order costs (they require additional cooling components, additional memory and controllers, larger power supplies etc). Secondly, they typically require significant energy resources, which are prohibitive in small portable systems. Thus, the use of FPGAs is becoming attractive. In Section 4.3, a particular application that we consider representative, key point detection and tracking, is presented.

4.2. Sample application: Network Security and Intrusion Detection

(CONTRIBUTORS: Synelixis)

4.2.1. Application description

Network intrusion detection systems (NIDS) are widely adopted as high-speed and always-on network access demand more sophisticated packet processing and increased network security. Instead of checking only the header of incoming packets (as for example firewalls typically do), NIDS also scan the payload to detect suspicious contents. The latter are described as signatures or patterns and intrusion signature databases are made available that include known attacks. These databases are regularly updated and an NIDS has to be able to provide a certain degree of flexibility so that it can incorporate the updated security information.

In the past, NIDS used mostly static patterns to scan packet payload. Recently, regular expressions have also been adopted as a more efficient way to described hazardous contents. As such modern rulesets are comprised of both static patterns and regular expressions (for example the popular Snort IDS system upon Synelixis NIDS is based, includes more than 2000 static patterns and more than 500 complex regular expressions).
The general requirements of NIDS systems are (i) high processing throughput, (ii) low implementation cost (iii) flexibility in modifying and updating the content descriptions, and (iv) scalability as the number of the content descriptions increases. The performance requirements are fundamental to the correct functioning of an NIDS system, as if it cannot meet them, then the system itself is susceptible to specific types of attacks (overload and algorithmic attacks). Thus hardware-based NIDS systems are popular since they can provide the performance required.
NIDS systems implemented on reconfigurable hardware have the potential to marry the high performance of hardware-based NIDS systems with the flexibility of software solutions. Specific rules can be mapped to custom logic for maximum performance and changes (updates) to rules can be achieved by reconfiguring the device. Depending on the nature of the update (incremental versus more extensive ones), different reconfiguration approaches can be used: if a new rule is added in the system (the usual case), micro-reconfiguration can dedicate free resources to this new rule. If however, a major restructuring of the rules is performed (either as initial setup, a major upgrade, or to respond to new requirements of the organization) then partial or even full reconfiguration is needed.

4.2.2. Requirements

4.2.2.1 Computation Complexity

An analysis of the complexity of typical modern NIDS systems, such as the widely popular Snort[3] (upon which Synelixis NIDS system is based), reveals that by far the most complex task is the string matching process[4] and this is exactly the process that is most important to map on reconfigurable hardware. Recent rulesets of the Snort IDS express rules in both static patterns and regular expressions. The latter provide a more efficient way to represent hazardous packet payload contents and as a result their usage is becoming rapidly more widespread.
For static patterns, some form of the Boyer-Moore[5] algorithm is employed. For regular expressions pattern matching Synelixis uses a solution based on nondeterministic finite automata (NFA)[6]. Both computations have linear complexity regarding the length of the text string to examine and the length of the expression to be tested.

4.2.2.2 Type of Processing

Computation is mostly based on bitwise comparisons and decision tree traversals. Concerning an implementation in reconfigurable hardware, rather simple digital circuits are required to implement the related processes, mostly comprised of comparators, shifters and FSM logic. It should be noted however, that the way the algorithms work puts pressure in the memory subsystem and require the use of complex memory structures, such as CAMs.

[3] http://www.snort.org

[4] M. Fisk and G. Varghese, "An Analysis of Fast String Matching Applied to Content-based Forwarding and Intrusion Detection," in *Techical Report CS2001-0670*, University of California - San Diego, 2002

[5] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977

[6] Joao Bispo; Ioannis Sourdis; Joao M.P.Cardoso; Stamatis Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on* , vol., no., pp.119-126, Dec. 2006

4.2.2.3 Control Flow

Deterministic Finite Automata (DFAs) are typically used as control flow mechanisms for software implementations of the string matching processes, since sequential code is more suitable of keeping track of single active states. For a hardware implementation though, a DFA-based control structure can produce inefficient designs in terms of area (logic and memory). A worst case study[7] shows that a single regular expression of length *n* can be expressed as a DFA of $O(\Sigma^n)$ states. Synelixis approach is based on Nondeterministic Finite Automata (NFAs). Compared with a DFA, an NFA representation for the aforementioned case has only $O(n)$ states. The drawback is that an NFA has multiple active states, however since hardware is inherently concurrent, it is relatively easy to concurrently track those states.

4.2.2.4 Data Flow

NIDS is a streaming application. Data flow is linear as network packets pass through the NIDS system from a source to an end server. No data storage is performed nor is there any loop-back feed.

4.2.2.5 Parallelism

Parallelism can be exploited in two ways for an NIDS system. Different network packets can be processed individually; therefore if there are multiple feeds of packets (e.g. a central point with multiple network links) these can be processed simultaneously. On the other hand, string matching on a particular payload can be executed in parallel for all applicable rules.

4.2.2.6 Memory Constraints

The memory requirements of Synelixis NIDS system are directly related to the number of static patterns, as well as the static strings in the regular expressions included in the ruleset that has to be examined. Synelixis implementation requires a separate module for each regular expression. Using (partial or full) reconfiguration to update the ruleset means that each regular expression module can be hardwired to a specific circuit and no memory has to be dedicated to store control information in order to compute current or future regular expressions.

4.2.2.7 Targeted Performances

Performance is critical for an NIDS system. On one hand, the communication links offer higher and higher bandwidths putting pressure on the NIDS system as more packets per second have to be processed. On the other hand, evolving and more sophisticated attacks increase continuously the ruleset of an NIDS system, thus requiring more processing per packet.

It should be noted that the performance of an NIDS system is also directly related to the level of security it can offer. There are specific types of attacks, named overload attacks8, in which an attacker overloads an intrusion detection system by flooding it with innocuous packets until the system starts dropping packets. There is then a high probability that the detection system will not catch an attack that is interjected in this stream of packets.

To have strong assurance that a detection system is not subject to such attacks, the system must be able to support full utilization of the network that it is monitoring. As a result, the higher the performance (as measured in terms of throughput) that the NIDS system can offer, the greater the appeal of the solution to a broader market.

[7] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Reading, Mass.: 2nd Ed., Addison-Wesley, 2001

[8] Vern Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999

4.2.2.8 Reconfigurability

An NIDS system has to be updated in a number of different ways throughout its operation:

- Small incremental updates may be required to add, change or expand certain IP addresses or address ranges that appear in detection rules
- New static pattern rules may have to added to the static ruleset or changes to the current patterns included in the ruleset may have to applied
- Updates in the regular expressions to cover more cases or correct mishandling of certain detection rules
- New regular expressions may have to added
- Overall updates to the system may have to be performed in case of new policies or large-scale update to the operation of the NIDS system.

From the above, micro-reconfiguration methods appeal to the small incremental updates, as these appear to be the most frequent and require the smallest changes. This way, the system operation is practically not disrupted at all, while changes are applied almost instantly. Region-based reconfiguration seems more attractive for updates that require more significant circuit-level changes, such as significant updates or additions to the regular expression that are used to detect specific classes of patterns. Lastly, static partial or full reconfiguration may be required for system-level changes (as a result of a new policy for example).

4.3. Sample application: Image Analysis

(CONTRIBUTORS: ST)

4.3.1. Application description

STM is developing a demonstrator in the domain of image analysis for keypoint detection and tracking in order to address another promising domain, Augmented Reality. The detection of the feature points is done by the FAST (Features from Accelerated Segment Test) algorithm. The tracking itself is performed by PKLT (Pyramidal Kanada Lucas Tomasi).

4.3.2. Requirements

4.3.2.1 Computation Complexity

The corner detection step (FAST) needs to perform a high number of comparisons between pixel luminances inside a single image. The resulting features are passed to PKLT for tracking.

The tracking step (PKLT) is much more compute intensive. Filtered images need to be produced from the original frame, locale derivatives and gradient matrices must be compute per feature location. An iterative process is performed in order to compute the optical flow, based on a cascade of image at different level of details.

4.3.2.2 Type of Processing

The Image Analysis pipeline takes a video frame in input, identifies unique features in the image and track their locations between successive frames. This complex task is performed by two main stages implementing a corner detection algorithm (FAST) and a corner tracking algorithm (PKLT).

In the front-end, FAST processes each pixel of an input luminance image (single channel) for identifying potential corners, in comparing the luminance value of the pixel with its surrounding neighbors (located on a rasterized circle centered on the processed pixel). If a pixel is identified as possible corner, its associated score is computed according to the same neighborhood. This score represents its qualities as good corner compared to other candidates immediately surrounding it. Finally, a non-maximum suppression step occurs over all the corner candidates, suppressing potential corners adjoining another one with a better score. The final result consists in a list of corners, or features, which has to be tracked during the video sequence.

The back-end of the pipeline consists in the feature tracker, implemented by PKLT. The problem to solve here is to find the position of a corner in the current frame, knowing its position in the previous one. One solution is to compare a window in the current frame, to a window centered on the corner in the previous one. The translation that minimizes the difference between these two windows defines the optical flow, solution of the tracking phase.

The robustness and accuracy of this process depends on the size of the window considered. A large window offers a robust solution (for large movement for example), at the cost of precision. A small window increases the accuracy of the tracking, but may miss some correspondences. Consequently, a pyramid of images formed from the filtering of the full size frame at different resolution, offers a solution to this dilemma.

Hence, PKLT consists in producing the filtered cascade of input images, computing a spatial gradient matrix based on local derivatives around the corner to track at each resolution, and to perform an iterative Lucas-Kanade step for computing the local optical flow using the matrix previously computed. The final optical flow is the accumulation of the local optical flows at all the detail levels.

4.3.2.3 Control Flow

FAST consists in different level of loops, first across all the pixels of an input image, then across all the potential detected features, then again with data decimation.

PKLT loops across every feature detected by FAST. Two inner loops are processing these features, one covering the different level of details, while the last one covers each step of the iterative Kanade-Lucas processing for the current level of details. For each step, loops are used to compute gradient matrices and mismatch vectors over a given pixel window.

4.3.2.4 Data Flow

A stream of video frames is fed into FAST in order to detect features in these frames.

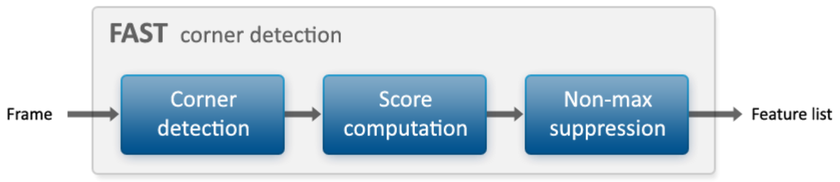

Figure 4 – FAST pipeline

A stream of feature list is sent to PKLT for tracking. A stream of optical flows is produced by this tracking stage.

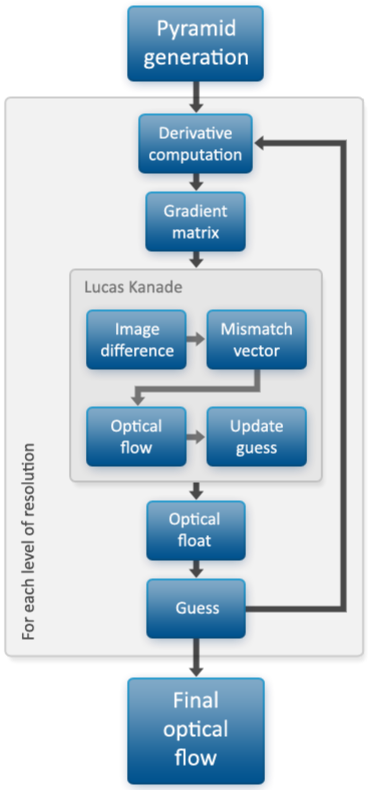

Figure 5 – PKLT pipeline

4.3.2.5 Parallelism

There are different levels of parallelism inside this pipeline.

The first level lies in the FAST front-end stage. Each pixel has to be compared to its neighbors for potentially being qualified as corner. As a consequence, this task can be performed in parallel for each pixel, independently. The same observation can be made for the score computation, following a similar process. However, the number of scores to compute is about two degrees of magnitude lower than the number of pixel to process (e.g. a VGA image is composed of 640x480 = 307200 pixels to process, with potentially ~1000 corners depending on the nature of the image itself).

The second level of parallelism lies in the PKLT back-end stage.

A cascade of images, at different level of details, has to be generated for the frame under process. A given level of details is computed according to the previous one and does not offer much parallelism. However, each pixel of a given level is produced by the filtering of the pixel of the previous level. This filtering process can be performed in parallel for each pixel, independently from its neighbor.

The gradient matrices and optical flows can be computed in parallel for each featured tracked by the algorithm.

4.3.2.6 Memory Constraints

Video frames are the most demanding in term of memory. FAST is using the luminance of a YUV input frame. PKLT is using the same frame, at different level of details (e.g. 4). For example, a VGA frame will demand 640x480 = 300Kb for the top level of details, or 400Kb for the complete pyramid.

4.3.2.7 Dependencies

The current version does not rely on any dependencies. Final release may be using OpenCL for explicit parallelization.

4.3.2.8 Targeted Performances

The performance targeted is real-time processing on an embedded system, for at least VGA resolution.

4.3.2.9 Reconfigurability

Some features of the Imaging Pipeline can be reconfigured.

The most obvious parameterization lies in the resolution of the input video stream. At minimum, a single channel VGA stream should be supported. With the availability of better and cheaper cameras, and increasing compute power, this resolution will increase certainly.

The frame rate of the input stream has also a direct impact on the performance of the application, ranging from 15Hz to 30Hz and above.

The FAST algorithm detects corners in comparing the intensity of pixels surrounding the evaluated point. The required number of consecutive pixels lower or above a given threshold has an influence on the number of corners, and their quality. Typically, a minimum of 9 consecutive pixels over a neighborhood of 16 is considered satisfying, but this limit can be increased in order to reach better results. This same observation applies to the choice of the threshold themselves, both in the corner detection and in the score phases.

During tracking, the PKLT algorithm works on windows of pixels (e.g. 9x9). The size of this window has a direct impact on the precision of the tracking, but also on its robustness. Consequently, depending on the application and the resolution of the input, different window size may be considered.

PKLT relies on different level of details of the input frame. For low resolution images (e.g. VGA), 4 levels of details is acceptable. However, with the increasing camera resolutions, the number of levels will also increase in order to keep a precise tracking.

Finally, the accuracy of the KL step performed by PKLT is controlled through an accuracy threshold, defined as an application parameter. The modification of this threshold has a direct impact on the number of KL iterations and consequently on the complexity of the computation and the final accuracy of the tracking.

5. Conclusion

Three application domains are addressed by FASTER, from HPC to desktop applications and embedded systems. These comprise a wide range of very different requirements, from system point of view (power consumption, available computational capabilities, memory, bandwidth, etc.) to application perspective (parallelization, execution times, data volume, computation complexity, etc.) and can benefit from different methods of reconfiguration in various ways.

As a representative of the HPC domain, Reverse Time Migration can use micro-reconfiguration to handle variations in domain size or in the number of timesteps for example. Partial region-based reconfiguration could be used to accommodate the change from imaging to propagation computation, while full reconfiguration may be necessary only when the physical model or other less frequent changes are necessary.

Ray-tracing is a demanding desktop/workstation class application, which may employ different reconfiguration methods to achieve the desire performances and flexibility. Different levels of reconfiguration can handle the various adaptable aspects of the application: from modification of the parameterization, such as secondary ray depth or pixel sampling, to more important reconfiguration of the type of supported geometric primitives, to even the global acceleration structure or the shading models used by the rendering process itself. It may even be beneficial to support reconfigurable accuracy (precision) of the intersection computations.

From the embedded domain, a NIDS system implemented on FPGA, can use micro-reconfiguration in order to accommodate frequent small changes to the detection rules (usually associated with IP addresses), while for larger changes to the ruleset (such as new regular expressions) region-based reconfiguration is an optimal solution. Full reconfiguration may be employed in cases where significant changes to the operation of the NIDS system are requires, as for example the application of a new system policy. Similarly, Image Analysis, another representative application of the embedded domain, may profit of reconfigurability in order to efficiently handle variations in video data input or different sizes of window used during feature detection or tracking, etc.

These requirements will have to be factored in order to define a unique and suitable design flow, as described in the Deliverable D1.2 *“Requirements of FASTER models, methods and tools”*.