3D Tomography back-projection parallelization on Intel FPGAs using OpenCL

Maxime Martelli, Nicolas Gac, Alain Mérigot, Cyrille Enderli

Journal of Signal Processing Systems manuscript No.
(will be inserted by the editor)

3D Tomography back-projection parallelization on Intel FPGAs using OpenCL

Maxime MARTELLI · Nicolas GAC · Alain MÉRIGOT · Cyrille ENDERLI

Received: date / Accepted: date

Abstract This article deals with the evaluation of FPGAs resurgence for hardware acceleration applied to computed tomography on the back-projection operator used in iterative reconstruction algorithms. We focus our attention on the tools developed by FPGAs manufacturers, in particular the Intel FPGA SDK for OpenCL, that promises a new level of hardware abstraction from the developers perspective, allowing a software-like programming of FPGAs. Our first contribution is to propose an accurate memory benchmark, and we follow with an evaluation of different custom OpenCL implementations of the back-projection algorithm. With some clues on memory fetching and coalescing, we then further tune designs to improve performance. Finally, a comparison is made with GPU implementations, and a preliminary conclusion is drawn on FPGAs future for computed tomography.

Keywords High-Level Synthesis · FPGA · OpenCL · Tomography Reconstruction · GPU

1 Introduction

Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. More like a roadmap, the current tendency shows its pace seizing up, a fact officially acknowledged by the industry. It was expected that physical limitations would eventually block

Maxime MARTELLI · Nicolas GAC
Laboratoire des Signaux et Systèmes, CentraleSupélec, CNRS, Université Paris Sud, Université Paris-Saclay, 3, rue Joliot Curie, 91192 Gif sur Yvette, FRANCE

Maxime MARTELLI · Alain MÉRIGOT
Laboratoire des Systèmes et Applications des Technologies de l'Information et de l'Énergie, ENS Paris Saclay, CNRS, Université Paris Sud, Université Paris-Saclay, FRANCE

Maxime MARTELLI · Cyrille ENDERLI
Thales Systèmes Aéroportés S.A., Elancourt, FRANCE

the circuit miniaturization, and the Semiconductor Industry Association announced the effective end of Moore's law for 2021 [6]. Circuits miniaturization (up to 10 nanometres thin today) should soon tip over microprocessors from the realm of traditional physics to quantum physics, which governs the probability behaviour of atoms. This technological shift is due to happen at the start of next decade. Until then, traditional chip thickness is expected to reach a ceiling of around 7 nm, stabilizing a high-cadenced progression over the years.

The processor industry, which already dealt in the past with other issues regarding Moore's law, like the increase of heat generation correlated with miniaturization, is now facing what looks like a deadlock. However, by considering performance instead of circuit density, the law can be transcended, and a logical evolution is to rethink commonly used architectures. One possible solution is for future computer chips to rely on a granular hardware specialization with reconfigurable fabric at their core, allowing processors to offload specific processing to a suited architecture.

With this in mind, Field Programmable Gate Array (FPGA) is a key technology for the post Moore era. Since the creation of the first mask-programmed gate array with Motorola's XC157 in 1969, FPGAs are widely used for specific needs like embedded [13] and critical [28] systems. For the last ten years, data centers have dramatically increased, and with this expansion comes a new research problem regarding power consumption. This market is due to be the largest one for FPGA technology for decades thanks to their technological potential, and the two main FPGA manufacturers (Xilinx and Altera[1]) developed tools (called SDAccel and Intel FPGA SDK for OpenCL, respectively) for the use of FPGA as mainstream software co-processing architectures, in a similar approach as Graphics Processing Units (GPUs) manufacturers did with OpenCL and CUDA toolkits.

Their main advantage is, for programmers, usage of well-known software languages. OpenCL is commonly used for GPU programming, and Intel FPGA SDK for OpenCL claims to allow program optimization by having little to no FPGA experience. This assertion is a potential game changer in the computing field and the mainstream adoption of FPGAs is now at stake. Over the last few years, many research work focused on FPGA implementations with OpenCL in various fields like lossless data compression [9], stencil codes [17], but also on proposing some clues about OpenCL code tuning for FPGAs [27] [23]. In this context, our approach is focused on characterizing possible bottlenecks for FPGAs implementations, and memory optimization.

As a use case, we consider the back-projection algorithm used in iterative reconstruction [10]. Computed Tomography (CT) is used in multiple domains such as medical imaging [12], quality control [25], or even food characterization [26], and its primary mission is to reconstruct a 3D cartography of an object's parameter.

The reconstruction of a N^3 voxels (stands for VOlume piXEL) volume from N projections on the N^2 detectors plane of the X-ray tomograph, is a time-

[1] Now Intel FPGA, since Altera's acquisition in December 28^{th}, 2015

consuming task. For the last decades, the data size is continuously growing with currently N=512 for medical imaging) and N=2048 for Non-Destructive Testing. To get a reasonable computing time, hardware acceleration has been sought on several architectures : CPU (openMP, MPI), FPGA [16] [20] [12], multi-FPGA like the imageProX by Siemens [15] or specific processors like D-SP, IBM Cell [18] or GPU [29] (before CUDA). However, from the last decade, the many core architectures belonging to the General Purpose GPU (GPGPU) family have proved to be the most efficient hardware accelerators specially the ones designed by Nvidia [22] with CUDA facilities. After the rise of GPG-PU, architectures designed on FPGA with a much longer conception flow, handwritten VHDL and fixed-point arithmetic has been put aside by the tomography reconstruction community with, however, some attempts [19] [30] to put back in the saddle FPGA architectures. Current FPGA architectures with a growing number of floating point computing units (DSP) and improved HLS tools are worth to be reviewed as an accelerator for 3D tomography reconstruction.

Even though GPUs can solve problems with multiple dimensions and large-scale data thanks to their Single Instruction on Multiple Data (SIMD) architecture, there are some specific algorithmic limitations in tomography like memory bottlenecks that have been unlocked with FPGAs [20] [12]. Therefore, there is an interest to estimate if new FPGA improvements make them competitive relative to GPUs for Computed Tomography.

In this paper our contributions consist firstly in evaluating the impact of new FPGAs tools in tomography, and secondly, in assessing OpenCL code optimization from a software engineer's perspective on Intel FPGAs.

The remainder of this paper is organized as follows: in Section 2, we present the back-projection algorithm to be implemented via OpenCL on FPGAs. Then, we introduce in Section 3 relevant concepts of the Intel FPGA SD-K for OpenCL. Section 4 describes our memory benchmark, while Section 5 deals with the different kernel optimizations. Finally, results are provided and discussed in Section 6.

2 3D Tomography algorithm

Computed Tomography relies on the analysis of a known radiation stream through the considered object to recover said physical characteristic by reversing the matter transport equation [24]. An X-ray source (Fig.1) revolves around the φ axis at $z = constant$. Radiation emitted from it is attenuated depending on the object local density, and a two-dimensional sensor array records intensity values, for each elementary φ angle. Those values are stored in a 3D matrix along (u,v,φ) in what is called a sinogram $s_{CT}(u, v, \varphi)$. From these sinograms, 3D volume is reconstructed using analytic algorithms like filtered back-projection [11] or iterative algorithms [14]. Both methods use the back-projection operator which represents respectively 90% and 50% of

the computing time [12]. For iterative algorithms, a projector operator is also needed but its acceleration won't be investigated in this paper.

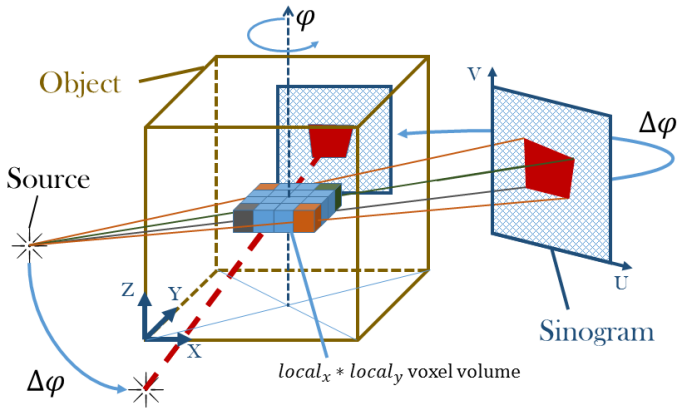

Fig. 1 3D Computed Tomography Projection.

The back-projection which equations are described in detail in [12] consists, for a given voxel $\mathbf{c} = (x, y, z)$, in summing up the contribution of every elementary detector (u, v) in line with the source and the considered voxel for every φ value. We then obtain the density $d(\mathbf{c})$ given as follows :

$$d(\mathbf{c}) = \int_0^{2\pi} s_{CT}(u(\varphi, \mathbf{c}), v(\varphi, \mathbf{c}), \varphi).w(\varphi, \mathbf{c})^2 \mathrm{d}\varphi \tag{1}$$

where $(u(\varphi_0, \mathbf{c}), v(\varphi_0, \mathbf{c}))$ are the values on the sinogram of the beam passing through $\mathbf{c}$ for $\varphi = \varphi_0$, and $w(\varphi_0, \mathbf{c})$ is the distance weight[21].

The sensors distribution being discrete, the integral transforms in a sum for all φ values. This algorithm is particularly suited for SIMD cores, because this sum has to be computed for every voxel of the object, and is best executed on massively parallel architectures. However, a way to accelerate our algorithm is to do coalesced memory access patterns, also referred as memory coalescing. Many times, memory objects are retrieved in large blocks, and cached in smaller but faster caches. To group contiguous memory accesses is an efficient way to improve the efficiency of our algorithm. Here, for a given voxel, the gathering of sinogram values for each φ iteration follows an irregular pattern, and a way to improve the algorithm is by grouping density computation per $local_x * local_y$ voxel rectangle to take advantage of memory coalescing. This particular point is discussed in Section 5.2.

3 OpenCL with the Intel FPGA SDK

3.1 Architecture

OpenCL is an abstract programming model and the corresponding basic architecture is showcased in Fig.2. Adapted to a co-processing implementation on heterogeneous architectures, the framework is designed to easily abstract most hardware considerations. As so, the host program is written in standard C, and communicates with Compute Devices via library routines that handle communication between the host processor and devices function, also called kernels.

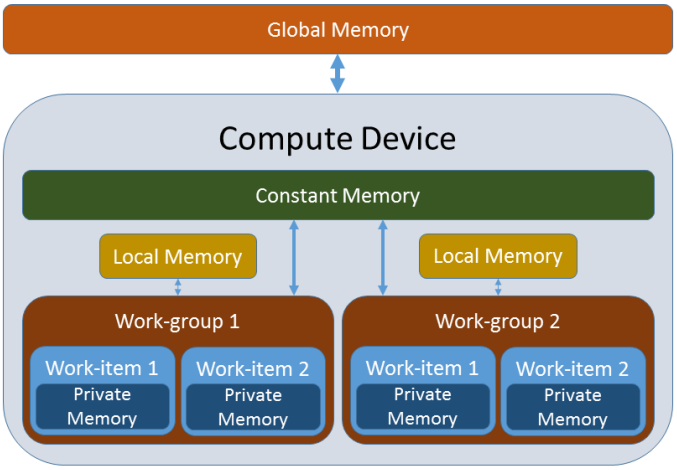

Fig. 2 OpenCL Memory architecture.

Each kernel instruction is transformed by the Altera Offline Compiler in a sequence of logic blocks, creating elementary pipelines that are then aggregated to form the kernel pipeline, which we refer to as Compute Unit (CU). There are two OpenCL kernel categories : NDRange (ND) and single work-item (SWI), detailed in Section 5. Generally, Intel FPGA SDK for OpenCL Guides [7] [8] recommends implementing a single work-item kernel in case of loop or memory dependencies, and a NDrange kernel otherwise.

Given the 3-dimensional back-projection problem, with a dimension of (dim_X, dim_Y, dim_Z) voxels, we can use the OpenCL work-group and work-item partitioning to handle its computation.

The 3D object is divided in elementary voxels in the three dimensions, and those voxels are gathered as shown in Fig. 3 in work-groups and work-items. In this case, a work-item, that is the elementary instantiation of a kernel, will do the computation of the volume density of a given (x_0, y_0, z_0) voxel, that is a sum of dim_φ sinogram values as explained in Section 2. Once the partition

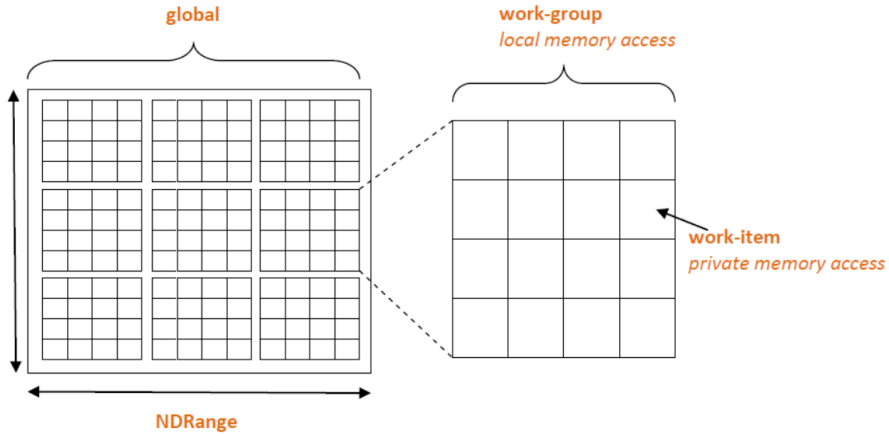

Fig. 3 OpenCL two dimension mapping. Source : engcore.com

is complete, the Altera Offline Compiler will implement as many pipelines as requested and is limited to the hardware resources available.

In Fig. 4, imagine an algorithm that needs to add five to each value of an array of 128 elements. We choose for this mono dimensional problem to split it in 8 work-groups, each containing 16 work-items. We therefore have 128 work-items in total, each in charge of handling the addition of one value of the array. In this example, we asked the tool to generate 3 processing units, allowing three work groups to be processed in parallel. Depending on the number of processing units, the Altera Offline Compiler will implement an enqueue mechanism to compute each work-group in an iterative pattern.

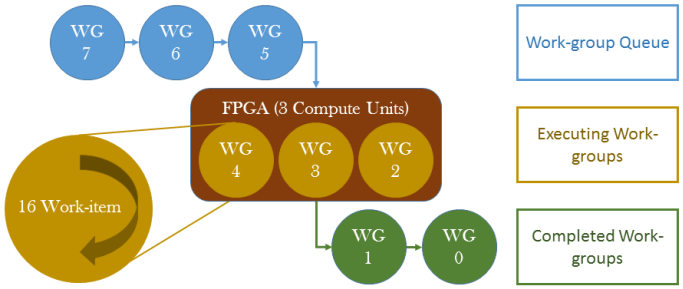

Fig. 4 OpenCL work-group enqueue mechanism.

3.2 Memory structures

As shown in Fig. 2, the OpenCL model has four memory types : global, constant, local, and private.

3.2.1 Global memory

Despite having the highest access latency amongst memory structures, global memory storage within the Intel FPGA SDK for OpenCL can still be efficient, thanks to automatic embedded on-chip caches implementation in Load-Store Units (LSUs) [7]. In case of repetitive global memory access, data is stored in embedded caches (direct-mapped 64 bytes cache with a 12 cycles mean latency) guaranteeing a high memory bandwith and a shorter latency compared to global memory, provided that memory access is not too large. A coalescent memory access through LSU embedded caches are the best way to optimize global memory bandwith.

3.2.2 Constant memory

Constant memory is implemented as on-chip global read-only memory. Access Port allocation size and width can be manually tuned to a maximum defined by the hardware manufacturer. However, latency improves substantially with the number of ports accessing the constant memory, thus simultaneous constant memory access per work-item must be narrowed to the strict minimum.

3.2.3 Local, and private memory

The main difference between local and private memory is their accessibility within a work-group. A private variable is stored in registers, and accessible only to one work-item, whereas a local one is visible to all work-items of a work-group. Altera Offline Compiler automatically implements local and private memories depending on the underlying access patterns, but users can also allocate and use those two memory types.

4 Memory characterization - custom benchmark

Manual memory handling is essential for effectively improving software implementation efficiency. In order to efficiently characterize each memory structure, we propose a custom benchmark to calculate their memory latency. The FPGA board to be tested is a DE1-SoC [2] coming with 1 GB of DDR3 memory and an Altera Cyclone V chip that integrates both a dual core ARM Cortex A9 processor and the FPGA fabric, with a maximum FPGA frequency of 305 MHz. All versions were compiled and synthesized using the Intel FPGA SDK for OpenCL 16.0. Measured execution time (Table 1 and Table 3) are obtained with the Intel FPGA Profiler tool.

4.1 Accurate measurements

4.1.1 Preventing the automatic embedded cache mechanism

As explained in Section 3.2.1, whenever we have a coalescent memory access, the compiler automatically implements LSU embedded caches. To prevent this, we can mark the data as "volatile" and the tool then ignores the memory optimization for the corresponding data. Even though it is the best way to optimize global and constant memory bandwidth, we want, to efficiently measure the different memory latency, to prevent the trigger of those caches.

4.1.2 Preventing automatic loop unrolling

Another FPGA-bound automatic optimization is loop unrolling. When it is possible, the compiler tries to unroll iterations in the inner loops when there is no data dependency and replicate the corresponding hardware to improve parallelism. For our benchmark, we do not want loops to be unrolled, and we implemented the randomization explained below in order to prevent it (Listing 1).

```
   /* Host randomization                                          */
   /* ...                                                        */
1  for int i = 0 to size - 1 do
2  |  host_in[i] = (int) (rand())  ( (float)RAND_MAX * size);
3  end
   /* Copy host_in to srcArray on device global memory           */
   /* Launching kernel                                           */
   /* ...                                                        */
   /* Kernel random pattern                                      */
4  int test_value = 0;
5  for int i = 0 to size - 1 do
6  |  test_value = srcArray[test_value];
7  end
8  outValue[0] = test_value;
```

Listing 1: Host array randomization and kernel random access pattern (For Global and Constant memory structure)

From the host side, we construct (lines 1 to 3) an array of *size* elements of random values between 0 and size - 1 (*host_in*). The kernel function then reads the first element of the array, and its value corresponds to the next element to read, and so on (lines 5 to 7). Eventually, the last read value is stored and recovered afterwards on the host. If we do not do this extra step, the tool will not launch the kernel because it detects that there is no output, and declares the function unnecessary.

Because each iteration in the kernel needs the previous iteration to know where to look in the array, the report explicitly states that data dependen-

cy was preventing loop unrolling, as expected. Therefore, we can accurately measure the real access time without automatic optimizations.

```
   /* Host randomization                                                    */
   /* ...                                                                  */
 1 for int i = 0 to √size - 1 do
 2 |  host_local_in[i] = (int) (rand())  ( (float)RAND_MAX * √size);
 3 end
   /* Copy host_local_in to globalArray on device global memory            */
   /* Launching kernel                                                     */
   /* ...                                                                  */
   /* Kernel random pattern                                                */
 4 async_work_group_copy(localArray, globalArray, √size);
 5 int test_value = 0;
 6 for int i = 0 to √size - 1 do
 7 |  for int i = 0 to √size - 1 do
 8 |  |  test_value = localArray[test_value];
 9 |  end
10 end
11 outValue[0] = test_value;
```

Listing 2: Host array randomization and kernel random access pattern (For Local and Private memory structure)

4.2 Getting past local memory size limitation

Each FPGA chip has a maximum allocable local memory size. On the DE1-SoC, we could not allocate large integer arrays ($> 1024^2$ elements) in local memory. Thus, we adapted the randomization explained above and implemented a specific algorithm for local memory benchmarking (Listing 2).

The idea previously mentioned in Section 4.1.2 is, to accurately measure the elementary access time, to average it over a large number of accesses. To compare all the memories access time, we need *size* memory accesses per different memory type. For local memory, instead of allocating an array of *size* elements and doing *size* reads (Listing 1), we allocate an array of $\sqrt{size}$ elements, and with two nested $\sqrt{size}$ loops (Listing 2, lines 6 to 10), it also results in *size* read. Lines 1 to 3 of the same Listing show the same mechanism as explained in Section 4.1.2 to prevent automatic loop unrolling.

Contrary to the global and constant memory which can be allocated directly from host before the execution of the kernel on the FPGA, the copy to local memory is carried out in the kernel. This new step translates in an additional variable for latency measurements, and is discussed and explained in the following section.

Table 1 Measured kernels execution time for x^2 memory accesses on an Altera Cyclone V.

Number of accesses	Global Kernel Time (ms)	Constant Kernel Time (ms)	Local Kernel Time (ms)	Private Kernel Time (ms)
$1024 * 1024$	1252.03	614.03	418.65	287.81
$2048 * 2048$	5007.85	1546.67	712.67	302.03
$3072 * 3072$	11267.56	3088.14	1193.67	343.13
$4096 * 4096$	20031.13	5253.55	1870.3	380.34

4.3 Proposed model for accurate latency measurements

All our kernels do *size* reads to their corresponding memory structure and one global write. Additionally, they have a delay due to the launch of the kernel by the tool. For the local kernel, we have in addition a copy of $\sqrt(size)$ elements from global to local memory. Our model (2) represents the total time of our different kernels in comparison with their implementation. Here, x represents the size of the array.

- ${R_\alpha}^2$: time of one read access (integer) to the α memory structure
- W_α : time including one write (integer) and the tool delay for launching kernels
- G_L : time to copy an integer from global to local device memory

$$\begin{aligned} G(x) &= R_G * x + W_G \\ C(x) &= R_C * x + W_C \\ L(x) &= R_L * x + G\_L * \sqrt{x} + W_L \\ P(x) &= R_P * x + W_P \end{aligned} \tag{2}$$

Our objective is to calculate the R_α coefficients, corresponding to the access time for one elementary read to the α structure, in order to compute the mean latency of the different memory structures.

4.4 Results and discussion

The goal of this benchmark is to efficiently measure the mean latency of the different memory structures. In order to do so, we measure the total execution time of the different kernels. Afterwards, we do a linear resolution of our system to achieve the smallest error variance. For the Global, Constant, and Private Kernel, the solution is a linear function obtained with the Least Square Method, whereas the Local kernel solution is a polynomial in $\sqrt(x)$ of the second degree, obtained through a polynomial interpolation in the Lagrange form. Measurements are shown in Table 1 for different array sizes, and the

[2] α = G,C,L, or P for Global, Constant, Local, and Private

solution of our systems is shown in Eq.(3) and superimposed to the measured values in Fig. 5.

$$\begin{aligned} G(x) &= 1.2 * 10^{-3} * x + 7.75 * 10^{-2} \\ C(x) &= 3.0 * 10^{-4} * x + 301.6 \\ L(x) &= 8.9 * 10^{-5} * x + 1.323 * 10^{-2} * \sqrt{x} + 318.1 \\ P(x) &= 6.4 * 10^{-6} * x + 280.5 \end{aligned} \tag{3}$$

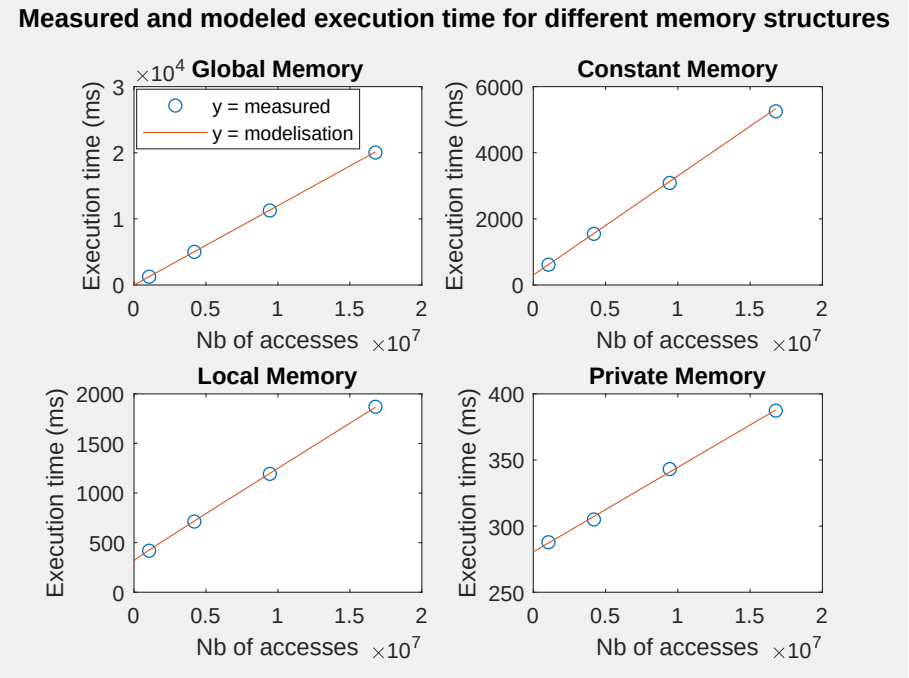

Fig. 5 Measured and modeled execution time for the different memory structures

This figure shows that our approach for the model is accurate, and we can therefore calculate the mean memory latency for the memory structures using Eq.(4). Final values are shown in Table 2.

$$Mean\ Latency(cycles) = Mean\ Read\ Time(s) * Kernel\ Frequency(Hz) \tag{4}$$

Our benchmark approach proved to be adequate, and the FPGA memory model for HLS is quite similar to GPUs, with the same principal characteristic : a reduced available size goes with a lesser latency. With this in mind, we could implement our different mechanism with efficient memory optimization.

Table 2 Measured memory latency on an Altera Cyclone V.

Memory structure	Kernel Frequency (MHz)	Mean latency (cycles)
Global	137.36	164
Constant	150.4	45
Local	137.11	12
Private	161.31	1

5 OpenCL 3D Back-projection implementations

The main FPGA advantage is its ability to be programmed for task or data parallelism. The Intel FPGA SDK for OpenCL allows both programming models, whose implementations are showcased further on.

5.1 Task parallelism - Single Work-Item

Single work-item is referred to as Task or Pipeline Parallelism on FPGAs [4]. Similar to the sequential model of a mono-threaded CPU program, there is no data repartition across work-items. In fact, Single work-item architecture is as in Fig.2, but with only one work-group and one work-item per work group. The Altera Offline Compiler optimization for this programming model is based on two core concepts : memory handling, and Shif-Register Pattern (SRP). This last concept is illustrated in Fig. 6. If an algorithm needs to access a memory object in a streaming pattern (the first work-item uses the first value, the second work-item uses the second, and so on), we can load this shared memory object in a ring counter and reduce memory latency.

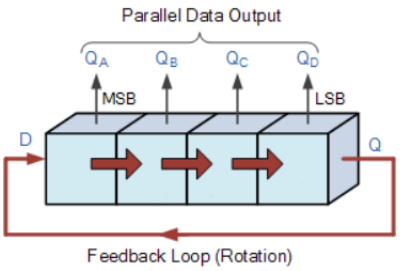

Fig. 6 Shift-Register Pattern - Ring counter. Source : electronics-tutorials.ws

Because the kernel is mono-threaded, the high-throughput is achieved by ensuring that at any moment, multiple instructions of the same kernel are processed concurrently at every pipeline step.

A key difficulty for single work-item implementations are loop handling, because the Altera Offline Compiler default behaviour is to have each loop iteration executed sequentially, thus drastically reducing the kernel throughput.

The baseline model for this article is the basic translation of Eq.(1) described in Section 2 as a single kernel on FPGAs, in a sequential CPU-like programming, and without any optimization.

From this implementation, a first optimization is to improve streaming throughput. By default, when a kernel needs to access an array, it allocates memory resources for efficient reads and writes. When an array access pattern matches with a streaming pattern, implementation can be modified to integrate a Shift-Register Pattern.

Input: $\alpha[dim_\varphi]$, $\beta[dim_\varphi]$, sinogram[dim_U*dim_V*dim_φ]
Output: volume, 3D array of reconstructed volume

```
1  local int2 SRP[dim_φ];
2  for φ = 0 to dim_φ - 1 do
3  |  SRP[φ] = (α[φ], β[φ]);
4  end
5  for zn = 0 to dim_Z - 1 do
6  |  for yn = 0 to dim_Y - 1 do
7  |  |  for xn = 0 to dim_X - 1 do
8  |  |  |  voxel_sum = 0;
9  |  |  |  #pragma unroll;
10 |  |  |  for φ = 0 to dim_φ - 1 do
11 |  |  |  |  SRP[dim_φ - 1] = SRP[0];
12 |  |  |  |  for i = 0 to dim_φ - 2 do
13 |  |  |  |  |  SRP[i] = SRP[i+1];
14 |  |  |  |  end
   |  |  |  |  /* Calculate (U_n, V_n) from SRP[φ]                    */
15 |  |  |  |  voxel_sum += sinogram[U_n, V_n, φ];
16 |  |  |  end
17 |  |  |  volume[xn,yn,zn] = voxel_sum;
18 |  |  end
19 |  end
20 end
```

Listing 3: Single Work-Item with Shift Register Pattern optimization

Listing 3 describes the back-projection algorithm explained in Section 2. The first step is loading in a local array the precalculated projection coefficient α and β (lines 2 to 4). For every voxel, they are sequentially accessed. To improve the efficiency of our implementation, we implement a Shift Register Pattern mechanism from lines 10 to 12. For every successive (x, y, z, φ) iteration, the implemented shift-register pattern shifts the data contained in the shift-register pattern array in a loop pattern. A streaming pipeline is generated through all the loops and shifted for each iteration, instead of a costly memory mechanism. This optimization allows the compiler to extract the parallelism between each loop iteration, effectively reducing the execution time as shown in the Single Work-Item with Shift Register Pattern SWI+SRP kernel version in Section 6.2.2. From a software developer perspective, those three lines seem counter-intuitive. In reality, the Altera Offline Compiler recognize

the shit-register pattern and implements it as a cascade of flip flops, sharing the same clock.

5.2 Data parallelism - NDRangeKernel

Data parallel implementations on OpenCL strongly depends on the underlying hardware architecture. GPUs have SIMD architectures, whereas FPGAs can be reprogrammed as such, but works best with pipelined architectures. As explained in Section 3.1, the Altera Offline Compiler instantiates an iterative loop in order to sequentially execute each work-group and, if the design allows it, multiple compute units can be implemented, allowing some work-groups to be processed in parallel.

Therefore, the main difference between data and task parallelism is the handling of shared local memory within a work-group. The shift-register pattern optimization (Section 5.1) of the projection coefficient α and β can no longer be implemented because the work-group repartition denies the pipeline streaming approach. The FPGA architecture inherent adequacy to pipeline parallelism does not necessarily mean all data parallel implementations are not efficient. One of the best advantages of data parallelism is allowing work-groups to share local memory objects, reducing memory stalls if memory handling is efficient. The main challenge of the back-projection algorithm being to access the sinogram array, we designed a custom pre-fetching algorithm suited for a SIMD architecture, and the implementation is discussed in the following section.

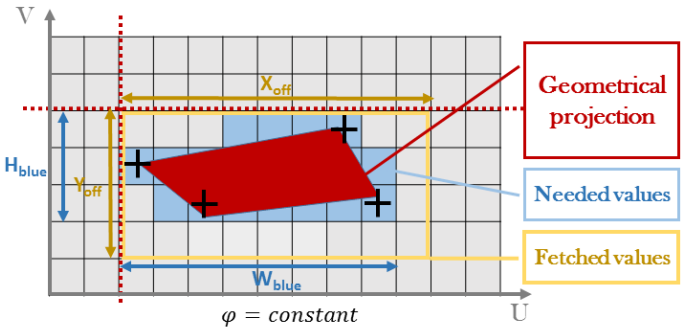

Fig. 7 Sinogram memory fetching pattern optimization.

As explained in Section 2, (U, V, φ) is the projection of a voxel over the detector matrix. For a given φ, a projection of a $(local_x, local_y, 1)$ voxel rectangle is alike the geometrical projection of Fig. 7, the four ends of the volume rectangle corresponding to the four black points. To compute the volume density of the initial voxel rectangle, the kernel needs to access all the cells of

the sinogram that are directly around the geometrical projection. They are the needed values of Fig. 7, and its access pattern cannot be predicted by the Altera Offline Compiler. Defining (W_{blue}, H_{blue}) as the Cartesian width and height of the needed values region, the geometry of the 3D back-projection problem guarantees the following :

$$W_{blue} < \sqrt{local_x^2 + local_y^2},\ H_{blue} <= 4 \tag{5}$$

In our implementation, $local_x = local_y = 16$. By choosing a local array dimension of $X_{off} * Y_{off} = 24*4$, we are assured to fetch all necessary sinogram cells needed for the work-group computation. Therefore, the implemented fetching algorithm (Listing 4) first calculates the top-left coordinates (Or bottom-right, depending on the array boundaries) of a matching rectangle (intersection of the dotted lines), and then evenly distributes the fetching of the values inside the constant size "Fetched values" rectangle. To guarantee a well-ordered execution, work-item synchronization is mandatory, and achieved at lines 4, 5, and 6.

Additionally, α and β arrays are stored in constant memory, in order to reduce memory latency through all work-groups.

Input: constant $\alpha[dim_\varphi]$, constant $\beta[dim_\varphi]$, sinogram[dim_U*dim_V*dim_φ]
Output: volume, 3D array of reconstructed volume

```
1 local int local_sinogram[X_off * Y_off];
  /* Recovery of work-item characteristics                          */
2 voxel_sum = 0;
3 for φ = 0 to dim_φ - 1 do
  |  /* Calculate U_n, V_n coordinates                              */
  |  /* Dispatch min, max coordinates computation between local work-items
  |     */
4 |  barrier(CLK_LOCAL_MEM_FENCE);
  |  /* Global sinogram fetching by local work-items               */
5 |  barrier(CLK_LOCAL_MEM_FENCE);
6 |  voxel_sum += local_sinogram[local_U_n,local_V_n];
7 end
8 volume[xn,yn,zn] = voxel_sum;
```

Listing 4: NDRange kernel fetching optimization

The drawback of this algorithm is that there are more memory fetching than needed, that is, in Fig. 7, all "Needed values" are always included in the "Fetched values" rectangle area, but its major advantage is that each line fetching is coalesced and shared between work-items of a work-group, thus improving burst read access and potentially reducing memory stalls.

6 Results and discussion

6.1 Experiment setup

In addition to the DE1-SoC board (see Section 4), the GPUs used for comparison are the NVIDIA Titan X [5] and the Jetson TX2 [3], both having the Pascal architecture, and the CUDA toolkit version is 8.0.

The considered volume is a 256^3 voxel cube, with 256 angles variations. Each kernel execution is monitored through the Altera OpenCL Profiler for the FPGA and NSIGHT for NVIDIA cards. For each implementation, these tools provide amongst other things the operating frequency, the execution time and memory stalls.

In the following subsection, we suggest a benchmark for this specific back-projection algorithm, to further discuss the execution results shown in Table 3. Kernel implementations match the different optimizations previously discussed in Section 5.

6.2 OpenCL optimization for FPGA

6.2.1 Performance benchmarking

The Altera Offline Compiler memory management reduces the maximum operating frequency for kernels, and use an irreducible percentage of logical elements for kernel enqueueing. To evaluate the minimum logical footprint of our algorithm, we construct the ND+Backbone kernel by removing all memory accesses in the code, with only the algorithm backbone remaining. This gives us two important information. Firstly, the logic utilization cannot be shrunk more than 21% on the DE1-SoC board, and secondly, the measured mean frequency for NDRange kernels is of 140 MHz.

Also, because of the specificity of each design, every kernel implemented uses a varying percentage of the total available logic elements. In order to accurately compare those designs, we propose a linear extrapolation of the execution time for a given logic utilization to a full-chip usage. To validate this assertion, we replicated the initial design (ND+Naive) into a design with two Compute Units (ND+2CU in Table 3) on the same chip. Each compute unit of this new kernel handles half of the total voxels. Results show that the time estimated with our linear model (Normalized Execution Time (NET)) is slightly slower than the actual replicated kernel execution time (ND+2CU Measured Execution Time (MET)). This is mostly due to the memory footprint of two compute units being smaller than twice the logic utilization used for a single one thanks to the Altera Offline Compiler optimizations. Overall, it validates our extrapolation as a good empirical model, and we can safely use the Normalized Execution Time to compare performances.

6.2.2 Task Parallelism or Data Parallelism ?

As discussed in Section 5.1, we implemented two different single work-item kernels. The first one (SWI+Naive) is the baseline algorithm for this article with no optimization, and the second one implements a shift-register pattern to reduce memory footprint and increase streaming efficiency. This simple improvement gives a normalized 4.5 speedup, underlining the importance of a comprehensive approach to the algorithm data access pattern for performance optimization.

With a single work-item there is no notion of shared memory within a work-group. Therefore, it has less logical footprint than a NDrange kernel. However, with an execution time of 67.5 s, the kernel mean frequency is of 63.6 MHz, compared to a maximum operating frequency of 140 MHz per normalized stream. Improving a single work-item kernel is closely related to optimizing memory handling and data streaming effectiveness, in order to increase kernel frequency.

The ND+Naive kernel is the GPU-like version of the back-projection algorithm, with no memory optimization. From this version, we implemented the replicated kernel (ND+2CU) already presented in Section 6.2.1 and the memory fetching kernel described in Section 5.2.

What is noticeable is that the Altera Offline Compiler top priority is to guarantee no kernel stall. With approximately an execution time of 30 s, and a 140 MHz operating frequency for all NDrange kernels with one normalized stream, this means that we execute one (x, y, z, φ) computation per clock cycle. Also, Normalized Execution Time for the memory fetched kernel (ND+MF) has a 1.4 speedup compared to the ND+Naive iteration, thanks to a reduced logical footprint.

From a software developer's perspective, the Altera Offline Compiler is quite effective. Firstly, a program can be implemented on FPGAs using two different kernel types suited for task or data parallelism, and this generic characteristic can be used over a wide range of algorithm implementations on FPGAs. Secondly, the automatic optimizations are focused on performance, and guarantees a filled pipeline. However, this automatization comes with two main drawbacks : a bigger memory footprint, and a reduced kernel frequency. Therefore, effective optimization tracks is to reduce the logical footprint and increase operating frequency, allowing more kernel replication.

6.3 GPU versus FPGA, consumption and performance

The DE1-SoC chip is a low-value product. For an adequate comparison to high-end GPUs, we compiled the ND+MF kernel with 17 replicated CU targeting the SX660 Arria 10 FPGA [1], using 98% of its logic elements. Within Quartus, we obtained the kernel operable frequency (260 MHz), and using the PowerPlay Early Power Estimator, we get the power of the design (2.27 W). Because we did not have access to a SX660 Arria 10 , we estimated the exe-

Table 3 Measured and Normalized Execution Time of various kernel optimizations on the Cyclone V SoC.

Kernel version	Logic utilization (%)	MET(s)	NET(s)
SWI+Naive	49	222.9	109.2
SWI+SRP	36	67.5	24.3
ND+Naive	55	32.26	17.7
ND+2CU	96	16.9	16.2
ND+MF	40	31.3	12.5
ND+Backbone	21	30.8	6.47

cution time by dividing the Measured Execution Time of the ND+MF kernel on a Cyclone V by the number of replication fitting on an Arria 10 SX660, and further multiplying it by the ratio of both designs operable frequency ($t_{Arria\ 10} = \frac{31.3}{17} * \frac{140}{260} = 0.991\ s$).

We compare the extrapolated execution time on an Arria 10 to the measured execution time measured on a Titan X Pascal GPU (11 TFLOPS) and on an embedded Jetson TX2 GPU (0.665 TFLOPS) optimized with the CUDA implementation from [12] adapted for cone beam geometry [11]. In terms of raw performance, FPGA is merely comparable to GPUs due, as discussed in Section 2 to the back-projection algorithm being appropriate for data parallel architectures. As shown in Table 4, even if an Arria 10 OpenCL implementation has a better performance per watt than on both GPUs, the same program is faster on a Jetson TX2 than on an Intel Arria 10 FPGA.

Table 4 Power and Energy consumption of the ray-driven back-projection best optimization on GPUs and FPGAs.

Device	Power (W)	Execution time (ms)	Energy for 256^4 voxel computation(mWh)
Titan X Pascal	250	12	0.83
Jetson TX2	15	253	1.054
Intel Arria 10	2.27	991	0.63

Even though performances were largely improved on FPGAs compared to the naive version, the inadequacy between algorithm and architecture remains a major obstacle for implementing this type of algorithm on FPGAs.

On the other hand, by observing the effectiveness of various architectures (Eq. (6), FPGA is much more effective than GPUs (Table 5). Indeed, the most optimized kernel on both FPGAs allows, at almost each clock tick, to update a voxel computation per elementary core.

$$Cycles\ Needed_{for\ one\ voxel\ update} = \frac{Kernel\ Time(s) * Freq.(Hz) * Cores}{Total\ updates\ of\ voxels} \tag{6}$$

Table 5 Performance comparative between FPGA and GPU for 256^4 voxel updates

Platform	Execution time (ms)	"Physical" cores	Frequency (MHz)	Cycles needed for one voxel update per core
GPU Titan X Pascal	12	3072	1417	12.16
GPU Jetson TX2	253	256	1300	19.6
FPGA Cyclone V	31300	1	140	1.04
FPGA Arria 10	991	17	260	1.02

FPGA with VHDL or Verilog is known for its efficiency (performance/watt). In our use case, this characteristic is preserved even with OpenCL. Indeed, its characteristic in VHDL is to be able to provide a fine-grained architecture fitted to the chosen algorithm, and to obtain a highly efficient design like in [12] or in [16], and, with the Intel FPGA SDK for OpenCL, we were able to harness this same efficiency with HLS. However, FPGAs low frequency and its lower number of Multiplication Accumulation floating units remain a limiting factor for acceleration compared to a GPU falling behind of a factor 5.

7 Conclusions and perspectives

In this paper we proposed an accurate characterization benchmark for memory latency measurement, and presented different FPGA optimizations using the Intel FPGA SDK for OpenCL. We achieved to port a CPU code on a FPGA, with an overall speedup of 8.74 between the naive and the best optimized kernel on a Cyclone V chip (respectively SWI+Naive and ND+MF[3] in Table 3). What first spring to mind is that the developer must be aware of its program specificity, and some hardware knowledge is required to fully harness the power of OpenCL on FPGAs. Even more, memory management is at the core of OpenCL implementation, and, on FPGAs, reducing kernels logical footprint is key for further optimizations. Despite those improvements, FPGA is lagging behind GPU implementations partly due to a mismatch between the algorithm and FPGA architecture. The back-projection algorithm showcased in this article is well suited for SIMD architectures whereas FPGAs are suited for pipeline-like designs. We observed that the algorithm backbone was using a significant percentage of the total available logic (ND+Backbone in Table 3)), and this overhead, caused by the OpenCL encapsulation, is not to be neglected. The Intel FPGA SDK for OpenCL, however, is an impressive tool, allowing our optimized FPGA HLS design to compete with hand-coded VHDL implementation in much less coding time (2 months in OpenCL vs more than a year for a 3D-cache memory fetching algorithm with VHDL [12]). Overall, it does a fine job constructing an adequate architecture for the algorithm, but eventually the lack of raw power from FPGAs compared to GPUs is a strong liability for massively parallels algorithms.

[3] ND+Backbone is, as explained in Section 6.2.1, not functional since all memory accesses are removed.

In this way, FPGAs resurgence for tomography is not due to happen unless manufacturers integrate dedicated graphical cores within FPGAs. However, their rise as a software-defined accelerator is promising. Through OpenCL, software developers can easily express the inherent data or task parallelism of their algorithms while exploiting a GPU-like memory model. In the near future, it is therefore viable to think about heterogeneous architectures where algorithms will be segmented depending on their inherent specificity, and each elementary partition executed on FPGA, GPU, or CPU devices, while using the same language: OpenCL.

References

1. Arria 10 Device Overview URL https://www.altera.com/en_US/pdfs/literature/hb/arria-10/arria_10_aib.pdf
2. DE1 SoC Device Overview URL http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836
3. Jetson TX2 specifications URL https://elinux.org/Jetson_TX2
4. OpenCL on FPGAs for GPU Programmers URL https://www.altera.com/en_US/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf
5. Titan X Pascal specifications URL https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications
6. The International Technology Roadmap For Semiconductors 2.0. Semiconductor Industry Association (2015)
7. Intel FPGA SDK for OpenCL Best Practices Guide. Intel (2017)
8. Intel FPGA SDK for OpenCL Programming Guide. Intel (2017)
9. Abdelfattah, M.S., Hagiescu, A., Singh, D.: Gzip on a chip : High Performance Lossless Data Compression on FPGAs using OpenCL. International Workshop on OpenCL (2014)
10. E.Kinahan, P., et al.: Emission tomography : the fundamentals of PET and SPECT, chapter Analytic image reconstruction methods. Elsevier Academic Press (2004)
11. Feldkamp, L.A., Davis, L.C., Kress, J.W.: Practical cone-beam algorithm. J. Opt. Soc. Am. A **1**(6), 612–619 (1984). DOI 10.1364/JOSAA.1.000612. URL http://josaa.osa.org/abstract.cfm?URI=josaa-1-6-612
12. Gac, N., Mancini, S., Desvignes, M., Houzet, D.: High Speed 3D Tomography on CPU, GPU, and FPGA. EURASIP journal on Embedded Systems (2008)
13. Garcia, P., Compton, K., Schulte, M., Blem, E., Fu, W.: An Overview of Reconfigurable Hardware in Embedded Systems. EURASIP Journal on Embedded Systems (2006)
14. Geyer, L.L., Schoepf, U.J., Meinel, F.G., Nance, J.W., Bastarrika, G., Leipsic, J.A., Paul, N.S., Rengo, M., Laghi, P.A., Cecco, C.N.D.: State of the Art: Iterative CT Reconstruction Techniques. Journal of Food Processing & Technology (2015)
15. Heigl, B., Kowarschik, M.: High-speed reconstruction for c-arm computed tomography. In: Proceedings of the 9th international meeting on fully three-dimensional image reconstruction in radiology and nuclear medicine, pp. 25–28 (2007)
16. Iain Goddard, M.T.: High-speed cone-beam reconstruction: an embedded systems approach (2002). DOI 10.1117/12.466946. URL http://dx.doi.org/10.1117/12.466946
17. Jia, Q., Zhou, H.: Tuning Stencil Codes in OpenCL for FPGAs. International Conference Computer Design (2016)
18. Kachelrie, M., Knaup, M., Bockenbach, O.: Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware. Medical Physics **34**(4), 1474–1486 (2007). DOI 10.1118/1.2710328. URL http://dx.doi.org/10.1118/1.2710328
19. Kim, J.K., Fessler, J.A., Zhang, Z.: Forward-projection architecture for fast iterative image reconstruction in x-ray CT. IEEE Trans. Signal Processing **60**(10), 5508–5518 (2012). DOI 10.1109/TSP.2012.2208636. URL https://doi.org/10.1109/TSP.2012.2208636

20. Leeser, M., et al.: Parallel-beam backprojection: an FPGA implementation optimized for medical imaging. VLSI SIgnal Processing Systems **39**(3), 295–311 (2005)
21. Lu, H., Cheng, J.H., Han, G., Li, L., Liang, Z.: A 3D distance-weighted Wiener filter for Poisson noise reduction in sinogram space for SPECT imaging. Medical Imaging, Physics of Medical Imaging (2001)
22. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast gpu-based ct reconstruction using the common unified device architecture (cuda). In: 2007 IEEE Nuclear Science Symposium Conference Record, vol. 6, pp. 4464–4466 (2007). DOI 10.1109/NSSMIC.2007.4437102
23. Shagrithaya, K., Kepa, K., Athanas, P.: Enabling Development of OpenCL Applications on FPGA platforms. Conference on Application-Specific Systems, Architectures and Processors (2013)
24. Thurston, M., Nadrljanski, M.M., et al.: Computed tomography - radiology reference article. Radiopedia
25. Vasilev, S.L., Artemev, A.V., Bakulin, V.N., Yurgenson, S.A.: Testing loaded samples using X-ray computed tomography. Russian Journal of Nondestructive Testing (2016)
26. Vidhya, M., Varadharaju, N., Kennedy, Z.J., Amirtham, D., Jesudas, D.M.: Applications of X-Ray Computed Tomography in Food Processing. RSNA (2015)
27. Wang, Z., He, B., Zhang, W., Jiang, S.: A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs. IEEE International Symposium on High Performance Computer Architecture (HPCA) pp. 114–125 (2016)
28. Wegrzyn, M.: FPGA-Based Logic Controllers for Safety Critical Systems. IFAC Conference on New Technologies for Computer Control (2001)
29. Xu, F., Mueller, K.: Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. Nuclear Science, IEEE Transactions on **52**(3), 654–663 (2005). URL http://cvc.cs.stonybrook.edu/Publications/2005/XM05
30. Xu, J., Subramanian, N., Alessio, A., Hauck, S.: Impulse c vs. vhdl for accelerating tomographic reconstruction. In: Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '10, pp. 171–174. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/FCCM.2010.33. URL http://dx.doi.org/10.1109/FCCM.2010.33