An OpenCL pipeline implementation on Intel FPGA for 3D backprojection

Daouda Diakite, Maxime Martelli, Nicolas Gac

An OpenCL pipeline implementation on Intel FPGA for 3D backprojection

Daouda Diakite[1], Maxime Martelli[1,2], and Nicolas Gac[1]
[1] Université Paris-Saclay, CNRS, CentraleSupélec, L2S, 91190, Gif-sur-Yvette, France.
[2] Thales DMS, Elancourt, France.

***Abstract*—3D back-projector computation is a time-consuming task, and hardware accelerators are used in order to speedup this algorithm. We propose a pipeline implementation of the 3D back-projection algorithm on a high-end FPGA using Intel FPGA SDK for OpenCL while presenting some optimization metrics for task parallelism. Compared to a non-optimized version on Arria 10, we achieved a speedup of 23 regarding execution time, by applying these techniques properly. We then compared these results with a low-end FPGA, CPU and GPU in terms of execution time and energy efficiency.**

***Index Terms*—Algorithm architecture co-design, Intel FPGA SDK for OpenCL, hardware acceleration, FPGA, Computed Tomography.**

I. Introduction

RECENT improvements of Field Programmable Gate Arrays (FPGAs) are increasingly attracting interest in the field of High Performance Computing. This is in part due to the multiplication of floating point computing units inside those architectures (DSP) and the improvement of High-Level Synthesis (HLS) tools which allow developers to generate a hardware implementation on FPGAs using software programming languages like C, C++ or OpenCL. One of the promises of those tools is to make FPGA development accessible by software engineers. Like GPUs or multi-core CPUs, FPGAs are the often considered and included in heterogeneous systems, in order to speed up time-consuming applications. One main advantage is to also be more energy efficient compared to GPUs and CPUs.

Some recent work focused on exploration of HLS tools efficiency includes [1], [2], [3] for Intel OpenCL SDK, or [4], [5] for Xilinx Vivado HLS. In particular, [6] presented an OpenCL implementation that was better than a Verilog/VHDL design in terms of frequency, latency, and resource utilization. The authors of [7] proposed a novel method which separates memory access from computation for OpenCL kernels based on a Low-Level Virtual Machine (LLVM) compiler. The authors of [8] use OpenMP to generate OpenCL code from a higher abstraction level, which is then synthesized by OpenCL SDK for bitstream generation. These works underline the interest for FPGAs in the High Performance Computing (HPC) field.

In the past years, FPGAs for Computed Tomography (CT) were configured with Hardware Description Language (HDL) [9], [10], [11], where the goal was to model a pipeline (from the algorithm) and perform a voxel update at every clock cycle. The main concern of the HDL implementation is the very long development time. More recently, new approaches based on HLS tools for FPGAs where developed, to obtain better reconstruction times and energy efficiency while accelerating development time. One such work proposed a hardware acceleration based-FPGA of the Maximum Likelihood Expectation Maximization (MLEM) algorithm [4] and achieved a significant acceleration in execution time compared to an optimised CPU multi-core version.

FPGAs are re-configurable, and this allows programmers to have a better algorithm architecture adequacy, by leveraging data or task parallelism. Furthermore, tools included in the Intel FPGA SDK for OpenCL allow a developer to implement a wide range of optimizations quickly, but give less control compared to the traditional HDL approach.

In this paper we propose a pipeline implementation of the 3D back-projection algorithm based on the Reflex Attila Arria 10 device, using Intel FPGA SDK for OpenCL 18.1. From a first implementation of this CT algorithm on a low-end DE1-SoC board [1], we show in this paper the benefits of using a higher-end FPGA and explore new optimization possibilities based on task parallelism.

The remainder of this paper is organized as follows: in Section II, we present the back-projection algorithm. Section III introduces some OpenCL metrics relevant to pipeline parallelism and Section IV details different optimization techniques. The results are discussed in section V and we conclude this work.

II. 3D computed tomography reconstruction

3D tomography used in medical imaging or non-destructive testing (NDT) aims to acquire the internal structure of 3D objects from external measurements. An object (3D volume) is placed between an X-ray source and an array of detectors as illustrated in Fig. 1. Detectors and the source rotate around the object and the radiation emitted from the source is attenuated depending on the object local density. A sinogram is created by stacking all of the measured attenuation from the detectors acquired at different angles.

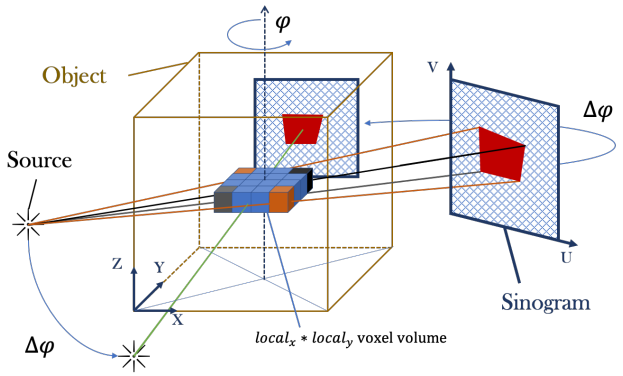

Fig. 1: 3D Computed Tomography

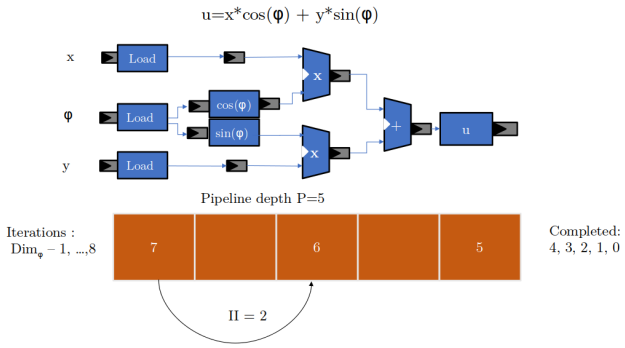

Fig. 2: Pipeline fragmentation of a function on FPGA

The 3D back-projection described in detail in [9] algorithm is given by

$$d(c) = \int S_{CT}(u(\varphi,c), v(\varphi,c), \varphi).w(\varphi,c)^2 d\varphi$$

Where (u, v) are the cone bean coordinates, φ is the angular trajectory of the detector and w is the distance weight.

$$u(\varphi,c) = x * cos(\varphi) + y * sin(\varphi)$$

$$v(\varphi,c) = x * sin(sin\varphi) - y * cos(\varphi) + z$$

As the sensors distribution is 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, one way to accelerate our algorithm is to make use of coalesced memory access patterns, also referred to as memory coalescing. Many times, memory objects are retrieved in large blocks, and cached in smaller but faster caches.

III. OpenCL metrics for pipeline characterization

For a given program, instead of having to transform the source code into instructions that depends on the target processor, High-level tools will create an architecture made of elementary blocks for each algorithm. Figure 2 represents a computation architecture which corresponds to the calculation of the u and v coordinates for the 3D back-projector. All data goes through the pipeline and we can see that a new iteration starts every two clock cycles with a pipeline depth of 5 due to data dependency. The overall efficiency of this generated hardware is of only 50% in theory.

Three main metrics characterize a pipeline. First, the **Initiation Interval (II)** represents the number of clock cycles between the launch of consecutive loop iterations. This means that one loop iteration is launched every II hardware clock cycles. The ideal II value is 1, and the offline compiler attempts to achieve that value for a given loop whenever possible and it is achieved when there are no dependency or if the dependency is handled in one cycle. In most cases, internal iteration dependencies causes poor II value and it can be fixed with some optimization techniques, like shift register pattern. Secondly, the **Pipeline depth (P)** is the number of cycles required by a data to pass through the entire calculation pipeline i.e. the number of stages of the pipeline. For a given number of iterations n, the global cycle number of a pipeline can be computed with those two metrics : $cycles = P + II(n-1)$. By adding a third metric, the **operating frequency (F)**, it is possible to compute the execution time of a pipeline, as given here : $T = \frac{P+II(n-1)}{F}$.

Optimizing an OpenCL implementation is finding a balance between those three metrics.

IV. Pipeline parallelism

OpenCL on FPGA supports data parallel (NDRange) as well as task parallel (Single Work Item) programming models and this makes FPGAs useful for a wide range of applications.

NDRange kernel is the GPU-like data parallel computing model on FPGAs using OpenCL which aims to perform the same set of instructions on multiple elements of a memory object. Each instance of execution is called work-item which are grouped in work-group. In [1] we performed an NDRange implementation of 3D back-projection and presented the OpenCL memory architecture as well as the impact of their utilization.

Fig. 3: Back-projection algorithm pipeline architecture

As Intel FPGA SDK for OpenCL allows both data and task (pipeline) parallelism, the choice of a type of parallelism, based on the application, requires an adequate knowledge of both the application specificity and the tools. In our case, since FPGA resources do not allow a massive data parallelism, we chose the pipeline parallelism approach to implement the 3D back-projection algorithm. Even before HLS tools, FPGAs were well known for their

deep and efficient pipeline execution pattern, resulting in good performances and low latency. For instance in computed tomography, a highly efficient hardware architecture for forward projection in Computed Tomography based on Xilinx Virtex-5 FPGA was proposed in [11] using a floating-point to fixed-point conversion and a two level memory, for separable-footprint (SF) forward-projector. Some OpenCL optimization techniques were presented in previous work [1], such as memory architecture and memory pre-fetching, as well as the difference between NDRange kernel and Single Work Item (SWI) kernel. Here, we will present other SWI optimizations based. The 3D back-projector is divided in several pipeline stages as shown in Fig. 3, which correspond to the crucial steps of the algorithm, this function level pipelining and each block can be split at loop level.

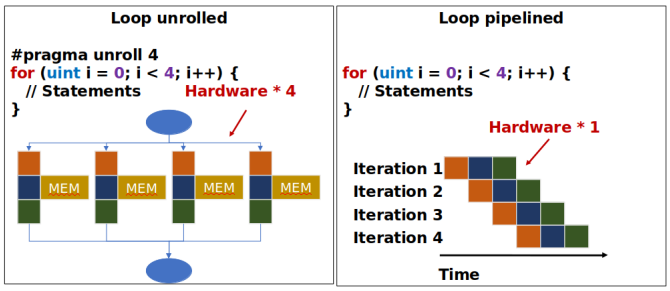

Fig. 4: Loop unroll vs loop pipeline

A. Loop pipelining

In SWI design, the offline compiler automatically tries to pipeline all the loops in the kernel. The idea is to split the body of the loop in a succession of micro-operations which can execute concurrently. The advantage is that the pipelined loop can launch a new iteration at every clock cycle by using the same hardware resources as the traditional sequential one but with better performances.

On FPGAs, the loop pipeline is the best solution in terms of resources and performances ratio. However an efficient pipeline must not stall which means it has to perform an ideal initiation interval.

In certain rare cases, achieving an II value of 1 might lower the kernel maximum operating frequency therefore lead to poor performances, for such a loop it is more convenient to use the ii pragma to fix the II value that performs the best frequency. It is also possible to fix the maximum operating frequency by using the F_{max} attribute or by including it in the aoc command at compile time. However, if the offline compiler cannot meet all the constraint bitstream generation will be impossible.

B. Loop unrolling

Another loop optimization is the unrolling, which consists of replicating, with a configurable factor, the body of the loop. Unlike pipelining, unrolling replicates also hardware resources which by the way maximize memory bandwidth utilization i.e maximally unrolling the most nested loop increases the amount of memory access per iteration, this is because the unrolled loop is closest to the computation. A loop can be fully or partially unrolled depending on the available logic element or the size of the cache memory (if applicable), the cache misses will increase if it cannot deliver all the data needed for computation.

V. Results and discussion

We used for this work the FPGA Attila Arria 10 device coming with 8 GB of DDR4 memory, with a maximum FPGA frequency of 405 MHz. The FPGA was in PCIe connection with the host system. The considered volume is a 256^3 voxel, with 256 angles variations. Each kernel execution is monitored through the Intel FPGA dynamic Profiler for OpenCL. For each kernel, this tool provides amongst other things the operating frequency, the execution time, the logic utilization, and the latency, bandwidth and stall of most memory access. The coefficient α and β are stored in the local memory due to their size and their frequent solicitation for each detector coordinates calculation.

In SWI kernel the loops being pipelined automatically the challenge is to feed the loop in the most efficient way, which brings us back to the loop unrolling. One of the problems in the case of a nested loop algorithm is the choice of the loop to be unrolled and the unrolling factor. One way to get to achieve the finest granularity possible is to first unroll the most nested loop and gradually consider the rest of the loops as long as the hardware resources allow. Based on this, in our case, the loop to be unrolled is the φ. This unrolling does not mean that several voxels are updated by clock cycle, but allows the accumulation of the sensor contributions to be accelerated according to the projection angles.

Board	HLS	Logic (%)	Freq.(Mhz)	time(s)
Arria 10	SWI φ unroll 1	24	196.97	124.82
Arria 10	SWI φ unroll 16	50	170.45	19.09
Arria 10	SWI φ unroll 32	62	150	5.34
Arria 10	SWI φ unroll 40	80	134.94	68.63
Arria 10	NDRange	86	168.23	28
DE1-Soc	SWI	36	63.6	67.5
DE1-Soc	NDRange	96	140	16.9

TABLE I: Performances comparaison between FPGAs Arria 10 (764k LUTs) and DE1-SoC (118k LUTs)

However, our first implementation, in table I, is the CPU-like version FPGA-friendly implemented to get an efficient pipeline with ideal II value and the *unroll*1 forces the compiler to not unroll the loop. The offline compiler achieves for this version an operating frequency of 196,97 Mhz with an execution time of 124,82 s. This is the result of under-utilization (only 24% of hardware resources In table I) of available resources on Arria 10 device and therefore it doesn't take full advantage of the parallelism potential. Exceeding this frequency by applying the optimizations, based on SWI, will be almost impossible as

the depth of the pipeline will be variable according to the number of replication.

By replicating 16 times (unroll 16), we double the number of resources and get a 6.5 fold acceleration compared to the first version. We notice that the offline compiler sacrificed F_{max} to achieve the II value of 1. The best execution time was obtained by a replication factor of 32 with 80% logic utilization at $150Mhz$, this version is the most optimal and takes advantage of automatic cache mechanism. Having the same features of this version with a larger frequency, using F_{max} attribute, was not possible for the offline compiler, it increased the number of constraints at the place and route time.

For this algorithm on Arria 10 device, the maximum possible replication factor is 40, once greater than 40 we exceed 100% of the device capabilities. This version is sub-optimal not only because of the low frequency but because the number of iterations is not divisible by 40. There will be an epilogue to handle which will substantially increase the depth of the pipeline and lead to poor performances. It is imperative, as far as possible, to know the number of iterations of a loop at compile time to have a bitstream that best fit the algorithm.

Algorithm 1 Kernel back-projector OpenCL

```
for all z_n, y_n, x_n do
  voxel_sum ← 0
  #pragma unroll factor
  for all φ do
    Compute(u_n, v_n)
    voxel_sum += sinogram[u_n, v_n, φ]
  end for
  volume[x_n, y_n, z_n] = voxel_sum
end for
```

A. FPGAs Arria 10 vs DE1-SoC

By comparing results shown in Table I with those obtained previously in [1], we observe that DE1-SoC SWI implementations were limited by its logical resources, as the tool was unable to generate an efficient pipeline kernel. Arria 10 implementations delivers better performance than both SWI and NDRange versions on DE1-SoC with 12.6 and 3.1 speedup respectively. It should be noted that interpolation was replaced by the nearest neighbor method on the DE1-SoC and the Arria 10 because it required more resources and computation. Besides, on DE1-SoC device the accumulation of a variable cost 6 clock cycle instead of one cycle for Arria 10. In terms of efficiency, the DE1-SoC is more efficient than the Arria 10 device when considering the number of logic elements.

B. FPGA, CPU, GPU

The GPUs are far ahead regarding performance (Table II) mostly thanks to their high core count and frequency, which makes them more suited for massive data parallel computing. However, FPGA implementation provides better execution time than an optimized multi-core CPU version despite its low operating frequency. Overall, GPUs still consume less energy than FPGAs despite requiring more instantaneous power.

Device	Power(W)	Energy(mWh)	Execution time(s)
CPU E5-2667	47	862	66
Titan X Pascal	237	0,92	0.014
Jetson TX2	12,9	0,91	0,253
Arria 10	9,9	14,65	5,34

TABLE II

VI. Conclusions

In this work, we presented a pipeline implementation of a 3D back-projector on an Intel FPGA Arria 10. FPGA achieved better performance than CPU but it does not perform as well as the optimized versions on GPUs. Considering the specifics of the FPGAs and metrics presented, it is possible for the offline compiler to generate a pipeline architecture similar, however less effective, to those generated through HDL languages.

In SWI implementation, parallelism is to be extracted in the body of the function and optimizations must be FPGA-centric to take full advantage of this architecture. This approach is recommended by the FPGA main manufacturers but if the application does not match this type of parallelism, it is a good idea to favor data parallelism. Tomography algorithms are better suited for massive data computing, which explains why GPUs are currently favored in this field.

To stand up to GPUs in tomography, FPGAs need to improve their ability to express data parallelism better like having adapted memory buses or more DSP.

We would like to thank Daniel Charlet from the LAL (Orsay, France) for granting us access to an Arria 10 board.

[1] M. Martelli *et al.*, "3D Tomography back-projection parallelization on Intel FPGAs using OpenCL," *Journal of Signal Processing Systems*, 2018. https://hal.archives-ouvertes.fr/hal-01831884
[2] A. A. Purkayastha *et al.*, "Exploring the efficiency of OpenCL pipe for hiding memory latency on cloud FPGAs," in *IEEE HPEC*, 2019.
[3] S. Hc *et al.*, "FPGA accelerated PET image reconstruction," *Engineering and Technology*, 2019.
[4] M. Ravi *et al.*, "FPGA as a hardware accelerator for computation intensive mlem medical image reconstruction," *IEEE Access*, 2019.
[5] F. Siddiqui *et al.*, "FPGA-based processor acceleration for image processing applications," *Journal of Imaging*, 2019.
[6] M. A. Mansoori *et al.*, "Efficient FPGA implementation of PCA algorithm for large data using hls," in *PRIME Conference*, 2019.
[7] A. A. Purkayastha *et al.*, "LLVM-based automation of memory decoupling for OpenCL applications on FPGAs," *MICPRO*, 2019.
[8] M. Knaust *et al.*, "OpenMP to FPGA offloading prototype using OpenCL SDK," in *IEEE IPDPS*, 2019.
[9] N. Gac *et al.*, "High Speed 3D Tomography on CPU, GPU, and FPGA," *EURASIP Emb Sys*, 2008.
[10] M. Leeser *et al.*, "Parallel-beam backprojection: an FPGA implementation optimized for medical imaging," ACM/SIGDA, 2002.
[11] J. K. Kim *et al.*, "Forward-Projection Architecture for Fast Iterative Image Reconstruction in CT," *IEEE Trans Sign Process*, 2012.