OpenCL FPGA Optimization guided by memory accesses and roofline model analysis applied to tomography acceleration

Daouda Diakite, Nicolas Gac, Maxime Martelli

OpenCL FPGA Optimization guided by memory accesses and roofline model analysis applied to tomography acceleration

Daouda Diakite[1], Nicolas Gac[1], Maxime Martelli[2]
[1] Université Paris-Saclay, CNRS, CentraleSupélec, L2S, 91190, Gif-sur-Yvette, France.
[2] Information Services Group, Inc.

***Abstract*—Backward projection is one of the most time-consuming steps in method-based iterative reconstruction computed tomography. The 3D back-projection memory access pattern is potentially enough regular to exploit efficiently the computation power of acceleration boards based on GPU or FPGA. This paper proposes an OpenCL acceleration of the voxel-driven 3D back-projection algorithm on an Arria 10 FPGA. This design flow is based initially on an offline memory access analysis, then iteratively on a performance analysis of each new implementation represented on a Berkeley Roofline model.**

By taking advantage of the FPGAs local memory architecture, we have succeeded to design an efficient pipeline reaching maximum bandwidth with stall-free access underlining this platform's interest for memory optimization. Our design flow allowed for a significant improvement of our initial algorithm's computational intensity, resulting in better performance on FPGA. It reaches comparable performance to an embedded GPU implementation and other computed tomography algorithms on FPGAs.

***Index Terms*—FPGA, HLS, memory access analysis, roofline model, tomography reconstruction**

I. Introduction

X-ray computed tomography(CT) is an imaging technique that initially found its application in the medical field. It has been extended to industrial applications such as non-invasive human body investigation and non-destructive testing of industrial materials. Model-Based Iterative Reconstruction (MBIR) algorithms are proved to produce better image quality at the cost of expensive computational time. To reduce the reconstruction time of CT algorithms, hardware accelerators are required. For the past few years, GPUs have been the preferred architecture due to their massively parallel computing pattern. However, FPGAs can be re-considered thanks to their low latency, power efficiency, and accessibility through High-Level Synthesis (HLS) tools provided by leading manufacturers like Intel or Xilinx.

Field programmable gate arrays (FPGAs) based on HLS tools are experiencing great consideration as an acceleration platform for many applications such as high-performance computing [1], [2], [3], deep neural networks [4], [5]. The maturity of their architectures and many built-in floating-point units (DSPs) in the latest FPGAs explain this interest. These floating-point units provide high design flexibility and are optimized to support high-performance DSP applications in IEEE 754 compliant floating-point single precision. For instance, Intel Stratix 10 [6] and Intel Agilex [7] devices can achieve up to 9 TFLOPS and 20 TFLOPS respectively. Unlike CPUs and GPUs, FPGAs can express spatial and temporal (fine/coarse-grained) parallelism, making them suitable for algorithms with sequential patterns and high data dependency. In the past, these various parallelisms are extracted for tomography through the HDL languages requiring a basic knowledge of hardware [8], [9], [10], [11]. This level of abstraction can be heavy and time-consuming development based on the complexity of specific algorithms. Hence, the emergence of tools with a high level of abstraction allows a broader audience to use FPGAs through software programming languages like C, C++, or OpenCL. FPGAs with HLS have recently been subject of evaluation in computed tomography algorithms such as Maximum Likelihood Expectation Maximization [12], [13], 3D back-projection [14], [15] or CT data alignment in memory [16]. These work focus on FPGA specific optimization for computed tomography in algorithm architecture co-design purpose. However, other works have based their optimization on the algorithm itself. Choi et al. [17] proposed a Ray-driven voxel-tile parallel approach hence take advantage of FPGA BRAM blocks and the data reuse rate. Zhang et al. [18] also proposed a parallel beam-based reconstruction on FPGA to exploit on-chip BRAM intensively.

In this paper, we propose an OpenCL acceleration of the voxel-driven 3D back-projection algorithm on an Arria 10 FPGA. This design flow is initially based on an offline memory access analysis then iteratively on a performance analysis represented on a Berkeley Roofline [19]. Our implementation is based on blocks of voxels reconstruction taking into account the date reuse rate, and exploiting the local memory on FPGA using Intel FPGA SDK for OpenCL. We compare the results with others FPGA implementations in terms of throughput, execution time, and design efficiency. The remainder of this paper

is organized as follows: in section II we present the 3D back-projection, the acceleration platform and the offline memory analysis. We introduce in section III our OpenCL architecture of the 3D back-projector. The optimisation guided by the roofline model and experimental results are provided in section IV.

II. Background and motivation

A. 3D back-projection algorithm

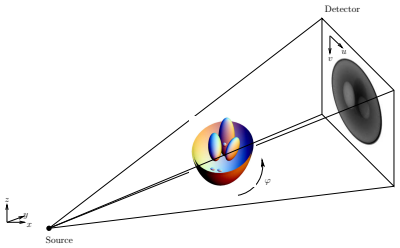

Fig. 1: X-RAY CT Projection

3D Computed Tomography (CT) aims to acquire the internal density d of 3D objects from external measurements S_{CT} called sinogram. An object (3D volume) is placed between an X-ray source and a detector plane as illustrated Fig. 1. The 3D back-projection used in iterative reconstruction and described in detail in [8] algorithm is given by:

$$d(c) = \int S_{CT}(u(\varphi, c), v(\varphi, c), \varphi).w(\varphi, c)^2 d\varphi \qquad (1)$$

where $c = (x, y, z)$ are the voxel coordinates, (u, v) are the cone beam coordinates, φ is the angular trajectory of the detector and w is the distance weight.

$$u(\varphi, c) = x * cos(\varphi) + y * sin(\varphi) \qquad (2)$$

$$v(\varphi, c) = x * sin(sin\varphi) - y * cos(\varphi) + z \qquad (3)$$

For each voxel (x, y, z), the projection of its contribution is located at a position $(u(x, y, \varphi), v(x, y, z, \varphi))$ on the detector. The contribution on the detector is computed by bi-linear interpolation. In our design, the interpolation is replaced by the nearest neighbor method to reduce resource consumption and computation overhead.

B. Acceleration platform

OpenCL is an open-source parallel programming API for heterogeneous processing platform (CPU, GPU, FPGA...). Based on the C99 standard, OpenCL supports both data and task-parallel programming models (Single Work-Item Kernel and NDRange Kernel) [20]. An OpenCL application is composed of two programs: a host application and the kernel compiled separately using Just In Time (JIT). The JIT compilation is not supported due to the long-time place and route step for bitstream generation on FPGAs. Therefore the OpenCL kernel is compiled offline using a vendor-specific compiler since FPGA does not support JIT compilation due to the place and route step in the synthesis flow. The Board Support Package (BSP) as shown in Fig. 2, provided by the board manufacturers, allows programmers to run the kernel executable on the target FPGA. It packages features such as IP Cores, DDR controller, PCIe controller, and DMA drivers to establish communication between the host and the FPGA device. Many Intel FPGA manufacturers provide FPGA with their BSP to quickly design and run Intel devices applications using Intel FPGA SDK for OpenCL. Intel FPGA SDK also provides, besides OpenCL directives, many FPGA-specific optimizations to fully harness the device potential.

These compilers implement different optimization based on their particular intermediate representation (IR), such as pipelining or expressing data-level parallelism. However, they suffer from a lack of support for many other unexploited FPGA-specific optimizations for arithmetic operations [21] for instance. Hence, harnessing FPGAs' full potential via HLS tools requires knowledge of their architecture and a significant effort to adapt the application because it is not performance-portable. This leads to the evaluation of OpenCL optimization techniques on FPGA by various works [2], [15], [22], [1].

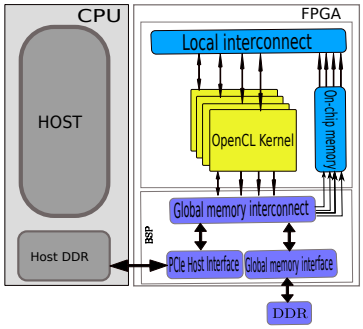

Fig. 2: Intel FPGA SDK for OpenCL platform

C. Offline Memory Access Analysis

The global memory access on Intel FPGA, despite its high latency, can be efficient for contiguous and repetitive memory accesses thanks to automatic embedded on-chip cache implementations in Load-Store Units (LSUs). For non-sequential and random accesses, these automatic caches, inferred by the Intel compiler, will be much less relevant to speed up the application on FPGAs efficiently. Their inference will be counterproductive and waste valuable BRAM resources with a high risk of memory stalling. Such behavior is confirmed in the 3D back-projection algorithm by [14], and many CT algorithms, so their acceleration on FPGAs remains a big concern. An offline study of the algorithm memory access pattern is required to make it regular or prefetching sinogram data to the on-chip BRAM before performing voxel reconstruction.

The projection data (sinogram) size is tremendous and cannot fit in FPGA on-chip memory. The block of voxels reconstruction will be wise to avoid global memory bottleneck and achieve better performance. The projection of a block (B_x, B_y, B_z) corresponds to a rectangle shape $(local_u, local_v)$ in the detector plane for a given projection angle φ_i. A high data re-utilization exists and is even more important for neighboring voxels. For each projection angle, voxels in the same block will access the same sinogram tile thanks to the CT system geometry. The main concern is to capture the sinogram footprint without loss of information and calculate the coordinates of its boundary. For each voxel (x, y, z), its reconstruction depends on its ϕ angular projections, these projections are spatially distant due to their storage in the sinogram following the order (u, v, φ). To increase the spatial locality, reconstruction by a group of voxels in the same block is beneficial compared to voxel by voxel reconstruction.

The prefetched sinogram data depends on the shape of the block of voxels. For a block (B_x, B_y, B_z), the 2D rectangle coordinates of the projection shape $(local_u, local_v)$ depend on B_x, B_y, and B_z such as: $local_v = \sqrt{B_x^2 + B_y^2}$ as pointed out in [15] and $local_v = \sqrt{2} * B_z$. The data reuse rate is computed by the following formula and illustrated in Fig. 3.

$Data\ reuse = \frac{B_x * B_y * B_z}{\#Memory\ access\ I/O}$ where the $\#Memory\ access\ I/O$ is obtained by a static analysis of the CPU code.

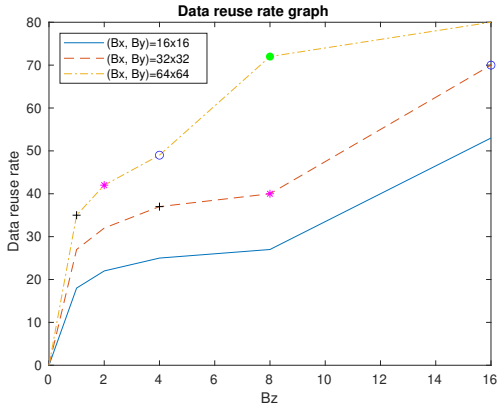

Fig. 3: Data reuse rate

The reuse rate is shown in Fig. 3 with different block sizes, same markers match the same number of voxels in the block, i.e. the same BRAM consumption. This allows us to see the effect of block shape and size that provides the best reuse rate. We notice that we have best trade-off from $B_z = 8$ (filled dot in Fig. 3) in data reuse and BRAM consumption.

III. Hardware implementation

The memory access pattern of the 3D back-projector is non-contiguous which could result in catastrophic performance on FPGA. The use of Intel automatic cache became a lock when optimizations such as loop unrolling were applied in the case of algorithm 1. Loop unrolling consists of fully or partially replicating the loop body and increasing the BRAM usage, thus preventing DSPs' maximum use.

Algorithm 1 Kernel OpenCL for BP-cache

```
for all z_n, y_n, x_n do
    voxel_sum ← 0
    #pragma unroll 32
    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
```

Algorithm 2 Kernel OpenCL for BP-prefetch

```
for all block_{z,y,x} do
    for all φ do
        Compute(u_0, v_0)
        Pretching sinogram data
        #pragma unroll 64
        for all z_n, y_n, x_n ∈ block_{z,y,x} do
            Compute(u_n, v_n)
            block[z_n, y_n, x_n] += sino_local[u_n, v_n]
        end for
    end for
    volume[z_n, y_n, x_n] ← block
end for
```

In our new implementation (algorithm 2), the critical path consists of reconstructing the block of voxels with the innermost loop over the voxels. The loop body, considered as processing element (PE), can be replicated for parallel voxel intensity computation by loop unrolling. It is then possible to have 64 PEs (see Fig. 4) in our architecture without exceeding available resources. Each PE must have free access to local memory, which requires a physical port for each memory read. If there is not enough port, the memory requests will be made with arbitration which will cause a severe performance problem of the pipeline by increasing the Initiation Interval (II) [23]. The architecture's input is the image projections in the detector plane as in Fig. 4. For each projection angle φ_i, we prefetch all contributions required (red rectangle) for the voxels accumulation in the block. We load more memory data than required to assure correct reconstruction and take advantage of memory coalescence. After the accumulation over all projections angle φ_i, the reconstructed block of voxels (blue cube) is written back (in the volume) to the global memory.

Our architecture requires 64 reads port (32 bits wide) for all the PEs to read sinogram data in local memory, and 2 read/write ports (2048 bits wide) for accumulation in the block of voxels. Intel recommends to have four or less read/write ports to local memory for stall-free access without arbitration [23]. We have a total memory replication factor of 22 to support parallel access to the local sinogram. Each replicate has the same sinogram data for the reconstruction. In addition, we place 8 privates

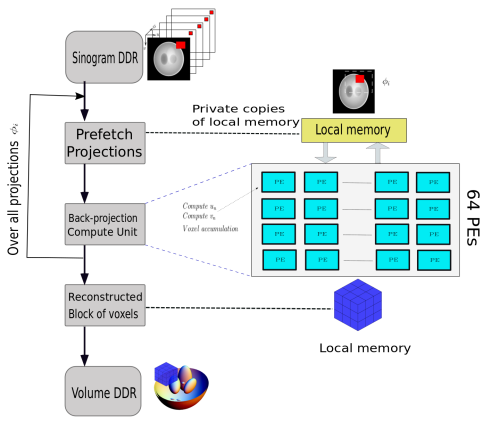

Fig. 4: BP-Prefetch architecture with 64 PEs

copies of the local memory for concurrent execution of loop iterations (φ loop in algorithm 2).

IV. Roofline analysis and results

The Berkeley Roofline model was used to highlight the optimization steps after improving our implementation's computational intensity. We first build our architecture to determine the attainable performance and then tuning the block size for better performance.

A. Experiment setup

In this experiment we do not consider the data transfer between the host and the device therefore the considered runtimes do not include memory transfer. However, to speed up the memory transfer, the allocate data must be at least 64 bytes aligned to allow the Direct Memory Access (DMA) transfer. To allocate an aligned memory, the posix function *posix_memalign* can be used in the host side. In our experience the aligned memory achieves better transfer rate than non-aligned in all cases. We used for this work the FPGA FLIK Arria 10 GX FPGA (10AX115N2F45E1SG) with 1150K logic elements, which comes with 8 GB of DDR4-2133 memory, with a maximum frequency of 480 MHz. The FPGA is connected in PCIe connection (via Thunderbolt 3) to 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.

B. Roofline model for FPGAs

The roofline model first introduced by [19] is a tool for visually and quickly observing the possible limitations of an algorithm relative to theoretical maximum performance on a target architecture. The model is characterised by two keys parameters which defines two roofs: the device peak performance and the attainable bandwidth. The work of Williams et al. [19], focused on CPU multi-core architectures, was extended to the FPGA architectures in [24] through HLS tools and taking into account the resource utilization of the device.

We use the roofline model to iteratively analyze our algorithm and guide the optimizations. Each algorithm results on a specific roofline for FPGAs. The performance roof is determined by the number of resources consumed and the effective operating frequency. The dynamic profiler gives the effective (measured) DRAM bandwidth achieved. Table I lists the number of operations GBOP and the number of global memory accesses of the 3D back-projection in GB.We determine the Computational Intensity (CI) for different versions of the algorithm. The BP-Cache design is memory-bound (see Fig. 5 red dot) due to the lack of spatial and temporal data locality in the sinogram. The computational intensity of this version is very low, elaborating another strategy to access off-chip memory might substantially increase the CI and allow the use of more DSP slices to improve the performance.

	B_z	Operations GBOP	DRAM (GB)	CI
BP-cache	N/A	236	17.2	13.7
BP-prefetch	1	253	1.1	236
	2	253	0.8	314
	4	253	0.67	377
	8	253	0.6	419
	16	253	0.57	443

TABLE I: Roofline analysis Memory and Operations

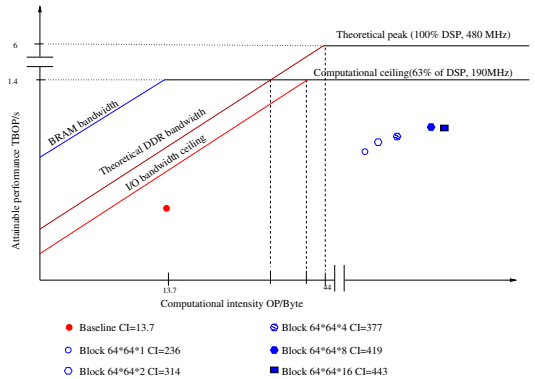

Fig. 5: Roofline of 3D back-projection with different B_z

We can see that CI increases as long as the block size grows until it reaches the maximum data reuse rate, and at the same time, the DRAM transactions is decreased. The memory operations in the table I take into account memory coalescence allowed by loop unrolling.

C. Effect block size variation

Table II shows the results of our implementation. The BP-cache design is a version of the 3D back-projection using burst-coalesced cached LSU (algorithm 1). OpenCL optimizations such as loop pipelining and unrolling were applied to this version to leverage the FPGA. This version

Version		BRAM (%)	DSP %	Stall (%)	Occ (%)	Freq (Mhz)	Time (s)
BP-Cache		72	27	71.27	24.6	150	3.65
BP-Prefetch	$32^2 \times 1$	50	58	88.34	12.3	197	2.78
	$32^2 \times 2$	50	58	90.1	24.3	190	1.46
	$32^2 \times 4$	50	58	7.22	43.5	195	0.893
	$32^2 \times 8$	50	58	0.44	60.3	180	0.719
	$64^2 \times 1$	50	63	2.79	43.5	183	0.843
	$64^2 \times 2$	50	63	0.34	58.9	186	0.658
	$64^2 \times 4$	50	63	0.2	74.4	187	0.529
	$64^2 \times 8$	62	63	0.06	84.1	189	0.425

TABLE II: Block size variation effect on the performance

suffers from a high pipeline stall percentage because of the memory access pattern making the global bandwidth the main bottleneck. The BP-Prefetch design (algorithm 2) achieves better performance compared to the BP-cache version. The table II shows the effect of block shape and size variation. We saw in Fig. 3 that the reuse rate varies with the number of voxels in the block. For instance, the blocks $64 \times 64 \times 1$ and $32 \times 32 \times 4$ have the same number of voxels and approximately the same reuse rate. However, the reconstruction based on the $64 \times 64 \times 1$ block is faster than the $32 \times 32 \times 4$. The memory access pattern is a bit different for the two block due to the data alignment. The performance for blocks with $B_x = B_y = 64$ are always better than $B_x = B_y = 32$ given the same number of voxels. In the design, all loops are pipelined with an Initiation Interval (II) of 1. Once B_x and B_y are fixed, B_z tuning allows to optimize the data reuse rate and therefore the overall performances as illustrated on table II. We have obtained a better execution time with the $64 \times 64 \times 8$ block, which corroborates the static study performed on the data reuse. Compared to the previous BP-cache design, we achieved a speedup of 8.6 at 188.9MHz.

The resource usage of the $64 \times 64 \times 8$ block version is presented in table III. As mentioned above our design contains 64 PEs, the overall back-projector design uses 949 DSP slices and 1952 blocks RAMs. The BRAM usage also includes the memory replication overhead to support concurrent access within the pipeline.

	FF	LUT	BRAM	DSP
Usage	407183	184616	1952	949
Available	1577720	788860	2537	1518
Ratio	25%	44%	62%	63%

TABLE III: FPGA resources consumption on Arria 10

D. Performance comparison

To fairly compare different implementations with different problem sizes, we use the Giga Updates Per Second (GUPS) indicator, which is insensitive to the size of the problem, by using the formula given in [25].

$$GUPS = \frac{GU}{Time_{kernel}} \quad with \quad GU = \frac{N_{voxel} * N_{acc/voxel}}{1024^3} \tag{4}$$

with N_{voxel} the size of volume and $N_{acc/voxel}$ the number of accumulation per voxel. The GUPS in [25] uses 1024^3 instead of 1000^3, so we use recalculate all GUPS according to this formula.

Our work achieved a comparable performance (same order of magnitude) to our embedded GPU implementation in terms of GUPS, as shown in table IV. For FPGA accelerations, Vivado HLS is commonly used to achieve acceleration as in [17], [16], although we use in this work Intel FPGA SDK for OpenCL. OpenCL is a bit at a higher abstraction than HLS. Hence the designer has more control over the pipeline using HLS compilers such as Vivado HLS or Intel HLS compiler. Therefore we have achieved low GUPS than the works using HLS. Moreover, we use much fewer resources than those works since our target device is a middle-end Intel FPGA. Choi et al. [17] used, with helical geometry, the Convey HC-1ex platform with four FPGAs running at 100 Mhz of operating frequency, and their design consumes 1408 DSP slices. Wen et al. [16] targeted the Xilinx ZCU102 platform (based on an UltraScale FPGA) with an overall DSP utilization of 1476 at 299.97 Mhz. We then evaluate all the FPGA implementations' design efficiency by comparing the Giga update perform per cycle by each MAC (Multiply ACcumulator) . Our OpenCL implementation on Arria 10 has approximately the same design efficiency as the HLS ones (table IV). We evaluate our pipeline efficiency by computing the update/cycle/PE, which is equals to 0.84 ; it demonstrates that our pipeline works efficiently close to the optimal compute throughput of 1 update/cycle/PE.

Reference	Platform	Volume	Acc. /voxel	Time (s)	GUPS	GUpdate /cycle /MAC
[17]	4×Virtex-6	$512^2 \times 372$	831	3.7	20.4	0.156
[16]	ZCU102	$1024^2 \times 128$	502	2.10	29.9	0.073
GPU	Jetson TX2	256^3	256	0.25	15.8	0.078
BP-cache	Arria 10	256^3	256	3.62	1.1	0.019
BP-prefetch	Arria 10	256^3	256	0.42	9.4	0.056

TABLE IV: Performance comparison of our work and other works

V. Conclusion

We present in this paper an optimization based on back-projection algorithm for CT reconstruction using FPGA BRAM efficiently. A reconstruction by block of voxels was developed to maximize data reuse and reduce external memory bandwidth, and maximize at the same time the use of the on-chip local memory. By expressing an higher CI for back-projection algorithm, our design implementation performs 9.4 GUPS based on an efficient pipeline with no stall percentage on Intel Arria 10. Performances are therefore becoming closer to the ones obtained on embedded GPU. We plan to extend this work to the ray-driven projector to run a full iterative reconstruction, and also to further optimize the implementations to target the Intel Stratix 10 GX device, which offers more computational capabilities.

References

[1] F. B. Muslim *et al.*, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," in *IEEE Access*, vol. 5, 2017, pp. 2747–2762.

[2] H. R. Zohouri *et al.*, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420, ISSN: 2167-4337.

[3] M. A. Mansoori *et al.*, "Efficient FPGA implementation of PCA algorithm for large data using hls," in *PRIME Conference*, 2019.

[4] E. Nurvitadhi *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. Association for Computing Machinery, 2017, pp. 5–14. https://doi.org/10.1145/3020078.3021740

[5] S. Zhang *et al.*, "Research on OpenCL optimization for FPGA deep learning application," in *PLoS ONE*, vol. 14, no. 10, 2019. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6786543/

[6] "Intel stratix 10," https://www.intel.fr/content/www/fr/fr/products/programmable/fpga/stratix-10.html.

[7] "Intel agilex," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/intel-agilex-fpgas-deliver-game-changing-com bination-wp.pdf.

[8] N. Gac *et al.*, "High Speed 3D Tomography on CPU, GPU, and FPGA," *EURASIP Emb Sys*, 2008.

[9] F. Pfanner *et al.*, "High performance parallel backprojection on FPGA," in *11th international meeting on Fully three-dimensional image reconstruction in radiology and nuclear medicine*, 2011. https://www.osti.gov/etdeweb/biblio/22124816

[10] M. Leeser *et al.*, "Parallel-beam backprojection: an FPGA implementation optimized for medical imaging," in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, 2002, p. 25.

[11] J. K. Kim *et al.*, "Forward-Projection Architecture for Fast Iterative Image Reconstruction in CT," *IEEE Trans Sign Process*, 2012.

[12] A. Cilardo, "Evaluating reconfigurable hardware for accelerating industrial CT," in *2020 IEEE 7th International Conference on Industrial Engineering and Applications (ICIEA)*, 2020, pp. 93–97.

[13] M. Ravi *et al.*, "FPGA as a hardware accelerator for computation intensive mlem medical image reconstruction," *IEEE Access*, 2019.

[14] D. Diakite *et al.*, "An OpenCL pipeline implementation on intel FPGA for 3D backprojection," in *6th International Conference on Image Formation in X-Ray Computed Tomography*, 2020. https://hal-centralesupelec.archives-ouvertes.fr/hal-02500994

[15] 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

[16] S. Wen *et al.*, "FPGA-accelerated automatic alignment for three-dimensional tomography," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 172–176, ISSN: 2576-2621.

[17] Y.-k. Choi *et al.*, "Acceleration of EM-based 3D CT reconstruction using FPGA," in *IEEE Transactions on Biomedical Circuits and Systems*, vol. 10, no. 3, 2016, pp. 754–767.

[18] W. Zhang *et al.*, "FPGA acceleration for 3d low-dose tomographic reconstruction," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020, pp. 1–1.

[19] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," in *Communications of the ACM*, vol. 52, no. 4, 2009, pp. 65–76. https://dl.acm.org/doi/10.1145/1498765.1498785

[20] K. O. W. Group, "The opencl specification: Version 1.2," https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[21] Y. Uguen *et al.*, "Application-specific arithmetic in high-level synthesis tools," in *ACM Transactions on Architecture and Code Optimization*, 2019. https://hal.archives-ouvertes.fr/hal-02423363

[22] K. Shata *et al.*, "Optimized implementation of OpenCL kernels on FPGAs," in *Journal of Systems Architecture*, vol. 97, 2019, pp. 491–505. http://www.sciencedirect.com/science/article/pii/S1383762118303151

[23] "Intel FPGA SDK for OpenCL pro edition: Best practices guide," p. 193.

[24] B. da Silva *et al.*, "Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools," in *International Journal of Reconfigurable Computing*, 2013, Research Article. https://www.hindawi.com/journals/ijrc/2013/428078/

[25] C.-Y. Chou *et al.*, "A fast forward projection using multithreads for multirays on GPUs in medical image reconstruction," *Journal of Medical Physics Research and Practice*, 2011.