# Real-time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability Trade-offs of the Altera OpenCL SDK

Quentin Gautier*, Alexandria Shearer*, Janarbek Matai*, Dustin Richmond*, Pingfan Meng* and Ryan Kastner*
*University of California, San Diego

***Abstract*—Embedding real-time 3D reconstruction of a scene from a low-cost depth sensor can improve the development of technologies in the domains of augmented reality, mobile robotics, and more. However, current implementations require a computer with a powerful GPU, which limits its prospective applications with low-power requirements. To implement low-power 3D reconstruction we embedded two prominent algorithms of 3D reconstruction (Iterative Closest Point and Volumetric Integration) on an Altera Stratix V FPGA by using the OpenCL language and the Altera OpenCL SDK. In this paper, we present our application and evaluation of the Altera tool in terms of performance, area, and programmability trade-offs. We have verified that OpenCL can be a viable method for developing FPGA applications by modifying an open-source version of the Microsoft KinectFusion project to run partially on a FPGA.**

## I. Introduction

Real-time 3D reconstruction creates a model of an object or environment by stitching together depth information from a camera or depth sensor at regular intervals. In 2011 Microsoft Research released KinectFusion which performs real-time 3D reconstruction with the movement of a Kinect around a scene. We believe that KinectFusion should be embedded to allow mobile devices to build applications atop this technology. We used FPGAs to embed the algorithms for 3D reconstruction to give us high performance and low power consumption, which are both critical for mobile devices.

The Altera OpenCL SDK allows a programmer to use high-level GPU code to generate an FPGA design with low-power consumption and good performance. We will use the Altera OpenCL SDK to compile GPU code from the open source KinectFusion (Kinfu) to FPGA designs [1]. The real-time constraints, high memory bandwidth requirements, and real world applicability of 3D reconstruction make this application a good choice for evaluating the Altera OpenCL SDK.

3D reconstruction is composed of three prominent algorithms. Ray-casting is a well-studied algorithm for generating graphics from surface information. The Iterative Closest Point algorithm (ICP) can be used for camera-tracking, a broadly applicable algorithm for graphics and vision applications. Volumetric Integration (VI) integrates depth streams into a single 3D surface and has not previously been embedded in an FPGA device. We are focused on embedding ICP and VI because they are the most intensive parts of the application. Our primary contributions for this paper are the following:

- Evaluating a new tool/method for producing FPGA designs and lowering the programmability barrier for embedding applications
- Embedding prominent algorithms on FPGAs used in 3D reconsruction
- Designing an end-to-end low-power 3D reconstruction heterogenous system which operates in real-time

Our evaluation of the Altera OpenCL SDK is informed by our experience in porting 3D reconstruction algorithms to FPGA while maintaining real-time performance. We will present how well the Altera compiler performs on the FPGA with the GPU code unaltered. Then we will walk through the iterations needed to tune our OpenCL code to generate the best performance on the FPGA. After we have described this process, we will present our final evaluation of the trade-offs in ease of programmability, performance, and area when using the Altera OpenCL SDK.

We describe related work in section II. Section III describes the 3D reconstruction algorithms. Section IV presents the FPGA designs for ICP and VI. In section V we describe the results and tradeoffs of our performance tuning. Section VI is a discussion of the Altera SDK and we conclude in section VII.

## II. Related work

The power and performance gains of using the Altera OpenCL SDK have been studied in [2]. They describe their use of the Altera tool for document filtering, an algorithm that has previously been embedded on and is well-suited to the FPGA. However, this work does not fully address the difficulties of porting a complex GPU application to the FPGA. Our work, in contrast, uses an application with challenging real-time performance and high memory bandwidth requirements, therefore it is a more realistic case study in evaluating the performance and code portability of the Altera tool and it adds more real-world value to the embedded application field.

An Iterative Closest Point algorithm on FPGA is presented in [3]. Their algorithm is optimized for object tracking. It uses a Region of Interest to filter the input data and a subsampling process to segment the tracked object. From the sampled data, they can run a type of nearest neighbor search to find corresponding points. However KinectFusion works on an entire scene instead of one object, therefore a ROI is impossible to define and this algorithm is not applicable. Also

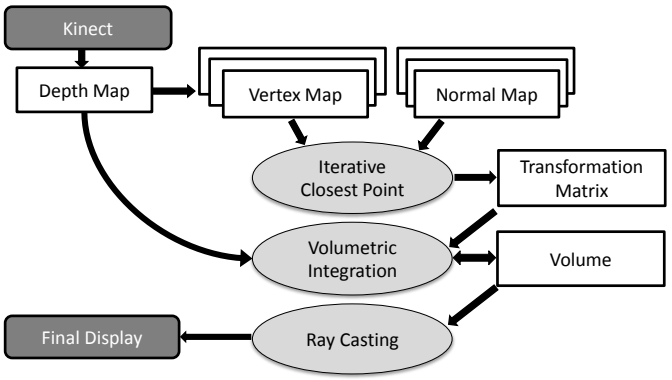


Fig. 1. General workflow of the Kinfu application. This shows the main algorithms and data flow between the Kinect and the final image displayed to the user. This entire process is repeated at most 30 times per second.

KinectFusion relies on dense tracking to enable features such as foreground detection based on ICP outliers. Because of this, ICP needs to run on all the original points, which makes a nearest neighbor search not feasible. For this reason we kept the original ICP algorithm from KinectFusion.

## III. 3D Reconstruction Algorithms

The general workflow of KinectFusion is similar to the one described in [4], [5] and presented in Figure 1. First the depth map is queried from the Kinect device and preprocessed. The processed depth map is then used to generate a cloud of 3D vertices with their normals (*Vertex Map* and *Normal Map*). These maps are down-scaled twice and used by the ICP algorithm to generate a transformation matrix, representing the movement of the camera between the current and previous frames. The VI algorithm uses this new location information with the original depth map to integrate with the existing 3D model (*Volume*). Then a Ray Casting algorithm computes the final image to be displayed on screen. This entire workflow is repeated up to 30 times a second.

The Iterative Closest Point algorithm we used is presented in [4], [5] and summarized in Figure 2. The input data are the vertex and normal maps of the current frame ($V_i$ and $N_i$) and of the previous frame ($V_{i-1}$ and $N_{i-1}$). Each point $p$ of the current map is projected into the previous camera coordinate space, then into image coordinate space. This image point $u$ is used as a lookup into the previous vertex and normal maps. The distance and angle between points $p$ and $u$ are then calculated and tested against threshold values to determine if they are corresponding or rejected as outliers. A point-to-plane error vector is calculated for each point that has a correspondence. These vectors are summed together to form a linear system, solved on the CPU to update the global transformation matrix between the frames. The initial transformation matrix is set to the one of the previous frame, then updated at each iteration. ICP operates on multiple resolutions of the input data (640x480, 320x240 and 160x120).

Volumetric Integration integrates depth information from a camera at regular intervals into a single representation of the volume [6]. In Kinfu the VI algorithm updates a volume of 512x512x512 32-bit integers, where each integer is composed of two values that represent the distance from a surface in the volume and the error in the distance calculation. Updating the volume requires the raw depth map from the Kinect sensor, which is an array of 680x480 16-bit values, and the transformation matrix and camera pose calculated from the ICP algorithm. VI iterates through each of the integers in the volume and updates the depth and weight values by interpolating with the new depth map if the voxels intersect with the camera pose.

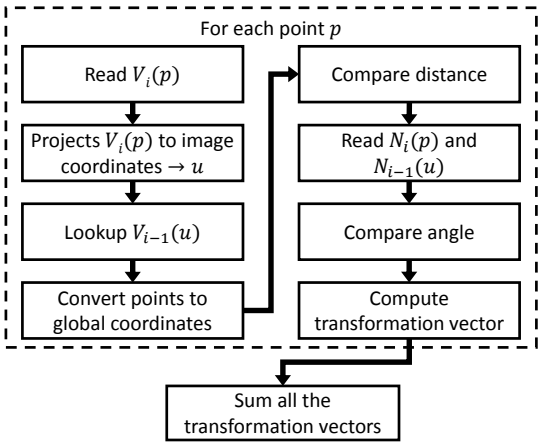


Fig. 2. This figure represents the various calculations performed for 1 iteration of ICP. $V_i$ and $N_i$ are the vertex and normal maps for the current frame. $V_{i-1}$ and $N_{i-1}$ are for the previous frame. The first part of the algorithm is a loop over all points from the input vertex map.

## IV. Implementation

We first ported the original CUDA implementations of each algorithm to OpenCL by keeping the same structures and features, with minimal differences. We consider that these OpenCL versions are the *baseline implementations*. Then, based on Altera's Optimization Guide [7], we made incremental modifications. We present the ICP, then VI modifications.

### A. Iterative Closest Point

*1) Baseline implementation:* The original GPU code for ICP is divided into the *search kernel* that calculates the correspondence between every points, and the *reduction kernel* that sums the transformation of all the corresponding points. The summation is done with a tree reduction that uses shared memory and synchronization between compute units.

*2) Kernel specialization:* Originally, both kernels ran a portion of the tree reduction, but we moved it entirely to the second kernel (*specialize kernels* in Figure 3). This created a bottleneck since the data had to be transferred through global memory.

*3) Loop unrolling and index dependency:* The original code used a double nested loop, where one index was function of the other. This resulted in poor optimization from the compiler. We removed the index dependencies and unrolled the loops manually (as it used slightly less board space than using OpenCL *pragma unroll*). Figure 3 presents the improvement of both modifications separately.

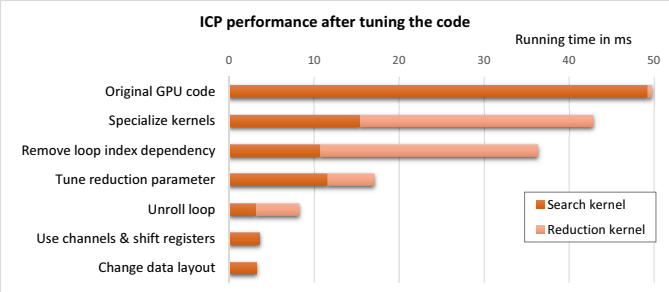


Fig. 3. Running time of 1 iteration of ICP at full resolution for the major modifications made to the baseline GPU implementation. The modifications are cumulative. The search kernel is the part of the algorithm that compares every vertex to find corresponding points. The reduction kernel sums the transformation vectors of all the points.

*4) Tune reduction parameter:* In the original GPU-style tree reduction, 512 work-items ran an individual summation of their own data, then a tree reduction was performed in local memory to get the final sum. We reduced the number of work-items to a number of 64 obtained experimentally to get much better performance. However we ultimately replaced the entire tree reduction as seen below.

*5) Altera channels and shift registers:* The GPU-style reduction was very inefficient on a FPGA, mainly because of the large data transfer through global memory. The access time and bandwidth limitation of this memory was slowing down both kernels. To remove this unnecessary access, we used Altera OpenCL *channels*. Channels are created in hardware as simple FIFO queues that can transfer data between kernels in 1 cycle per value. We used 27 of them to transfer the transformation vectors of 27 values to the reduction kernel. We tried to implement 27 independent reduction kernels, but found out that a single one was performing much better. We implemented it as an OpenCL task (single execution unit) to take advantage of loop pipelining provided by the compiler. To remove the inherent data dependency of a summation, we used a shift register. Shift registers have to be implemented manually by looping through the data, but this structure is recognized by the compiler and implemented efficiently in hardware. We used a 32 elements-wide shift register, and we chose the depth to be 8 values based on performance estimations. These techniques decreased the running time by 55%.

*6) Other optimizations:* a) We used compiler flags to optimize floating-point operations by reordering. b) We simplified the indexing of input arrays for the compiler to perform more aggressive optimizations. c) We changed the layout of the input data from Structure of Arrays to Array of Structures to take advantage of a 512-bits wide global memory access. d) We experimented with fixed-point arithmetic, however there is no native support in OpenCL for this format and this resulted in suboptimal design that did not fit on the FPGA board. Also, this implementation of ICP performs operations on both large and small numbers. This made it impractical to choose appropriate precisions for the integer and decimal parts without losing overall accuracy. e) We could not use SIMD operations or duplicated compute units because Altera channels cannot be vectorized, and because the ICP algorithm was using too much area on the board.

TABLE I. PERFORMANCE OF THE ICP ALGORITHM FOR 1 ITERATION

| | Running time | Area utilization |
|---|---|---|
| CPU (1 core) | 29.7 ms | |
| GPU | 1.33 ms | |
| FPGA (Baseline implementation) | 49.9 ms | 70% |
| FPGA (Tuned implementation) | 3.22 ms | 78% |

### B. Volumetric Integration

The main kernel the iterates through the volume and updates the values that represent areas that the camera pose intersects with the depth map. The distances from the surface as well as the weights are updated with a simple interpolation.

*1) Volume Access Restructuring:* The access pattern of the GPU code has work-items performing a maximum of 512 updates on the volume per frame. A for loop runs 512 times for each work-item. We have experimented with restructuring this code to allow 3D access and get rid of this for loop, which is okay for GPU but ruins FPGA performance. This has not yet been evaluated in terms of FPGA performance.

*2) Other Optimizations:* Since we could not restructure the volume access, we tried to unroll the loop. Unfortunately, unrolling the massive loop twice did not fit on the board. We used compiler flags to optimize floating-point operations and leveraged the Altera autotuner to perform combinations of the kernel parameters like number of compute-units and work group sizes, etc. The VI algorithm was also unable to have multiple compute units because of the presence of branching (if statements) in the kernel. Same with SIMD operations.

## V. RESULTS

First we discuss the performance of each algorithm running on its own, using a set of fixed input data. Then we discuss the integration of the tuned implementations into the complete Kinfu application. The FPGA benchmarks have been realized on an Altera DE5 board with a Stratix V FPGA, the GPU benchmarks on a NVidia GTX 760 and the CPU benchmarks on an Intel i7-4960X at 3.60 GHz.

*1) Iterative Closest Point:* Table I presents the running time of ICP for one iteration at full resolution on different platforms. The original code takes less than 1.4 ms on a GPU, but takes 49.9 ms on a FPGA. After tuning the code, we decreased the time to 3.22 ms on the FPGA with a clock frequency of 197 MHz. However we could not achieve one cycle per element because of several factors, including reads from unpredictable locations in global memory. We reduced the impact of these reads by grouping the elements, reducing the number of memory accesses, but it was not possible to make use of local memory since there is no data reuse. The execution pipeline is also delayed by multiple floating-point computations with dependencies, including divisions and square root functions. Another big limitation is the area utilization. Because ICP uses around 78% of the board, we could not integrate another algorithm on the same hardware.

*2) Volumetric Integration:* The baseline GPU implementation of VI ran on the FPGA in 100 ms, far from the GPU version which executed in 4 ms. Each iteration of the algorithm performs operations on 512 MB of data and there is little data reuse to optimize memory access. Transferring the data back

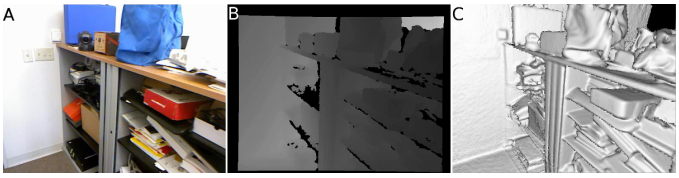


Fig. 4. Kinfu application modified to run ICP on the FPGA. The application runs at 26-28 FPS. (A) The Kinect RGB stream (not used). (B) The Kinect depth map. (C) The reconstructed 3D model of the scene.

and forth from the FPGA to the GPU would take too long and would require too much bandwidth (15 GB per second) to justify any peformance increases from the Altera compiler and the FPGA. The baseline version of volumetric integration used 89% of the board logic, which precluded placing the VI kernel on the same board as the ICP kernel.

*3) Integration with Kinfu:* We ran the Kinfu application on the GPU and FPGA descibed above. Due to the area limitation discussed before, we could only run one algorithm on the FPGA (VI or ICP) and the rest of the project on the GPU. For the VI algorithm, we realized that the data transfers between the GPU and the FPGA would take a significant portion of time, because the amount of data copied back to and from the FPGA totals over 512 MB. Given that, achieving a real-time solution for the resolution of 512x512x512 could not be accomplished in real-time. We are still investigating what is the maximal resolution we can support in our GPU-FPGA system so that can have a real-time application. For ICP, we had to copy the vertex and normal maps of two frames from the GPU memory to the FPGA off-chip memory. Even with aligned memory buffer on the host, this took around 18 ms and the entire ICP process 37.8 ms. The entire project ran at 13.5 FPS in average. To get a real-time performance, we removed the high-resolution input data and ran 12 iterations at 320x240 and 8 iterations at 160x120. The memory transfer took around 5 ms and the ICP algorithm ran in 13.5 ms with no noticeable accuracy loss. The entire system (ICP on FPGA, everything else on GPU) was running at 26-28 FPS (see Figure 4).

## VI. Discussion

*1) Altera OpenCL SDK:* We used the Altera SDK version 13.1. One problem we had with this tool is that we had to use most of the available area on the FPGA for only one algorithm in order to achieve real-time performance. One reason is the lack of fixed-point arithmetic support that makes it difficult to reduce the hardware utilization. Also, the compiler creates unnecessary logic that could be avoided with a manual RTL design. Finally, even though a good performance can be achieved with the Altera tool, it still lacks fine-grained control of the code to get the best performance and space utilization. But since this tool is still in its early stages, there is a good potential for improvement, and it will ultimately be a very useful tool for software programmers to make use of FPGAs without needing a deep knowledge of their technology.

*2) Iterative Closest Point:* By down-sampling the data for ICP to achieve real-time (see Section V), we may have compromised some KinectFusion features relying on ICP outliers. But if we could reduce the area utilization of ICP, we could run some preprocessing algorithms on the same board to get the input data directly in the FPGA memory and thus avoid expensive transfers. This operation would allow us to get the high-resolution data back in the ICP algorithm.

*3) Power:* By exporting a portion of the application to an FPGA, we can expect a lower power consumption. The GPU card that we used for our experiments has a peak power of 170 W and a peak core frequency of 1033 MHz. The maximum clock frequency of the FPGA for the ICP algorithm is 197.5 MHz, so we can expect a much lower dynamic power even with a high area utilization. Additionally the FPGA hardware is optimized to reduce both static and dynamic power. And if the Altera tool allows for more area optimizations in the future, then we will be able to run more algorithms on this hardware, including ray casting, drastically reducing the power consumption for each of these algorithms.

## VII. Conclusion

In this paper, we used the Altera OpenCL SDK to embed 3D reconstruction algorithms on an FPGA. We achieved real-time for Iterative Closest Point and used it to build an heterogeneous version of the open-source project Kinfu. For both algorithms however, we highly compromised the amount of area we used on the board and had to tediously optimize for the compiler, despite the promise of code portability that the Altera SDK offered. We showed that the Altera tool might still be an option to develop complex applications on FPGA as it improves over time and addresses some of its issues. We could also use the hardware designs and results with the Altera tool to inform our choices for developing a more space-efficient and powerful RTL design with both ICP and VI on a single board. Our evaluation of the Altera tool is valuable for the cause of moving towards cross-platform compatible code and applications, yet there is much more work to be done to realize the future of code reusability and performance portability across hardware platforms.

## References

[1] Point cloud library (kinfu project). Online: http://pointclouds.org/

[2] D. Chen and D. Singh, "Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 5–12.

[3] M. Belshaw and M. Greenspan, "A high speed iterative closest point tracker on an fpga platform," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, Sept 2009, pp. 1449–1456.

[4] S. Izadi *et al.*, "Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera." ACM Symposium on User Interface Software and Technology, October 2011. Online: http://research.microsoft.com/apps/pubs/default.aspx?id=155416

[5] R. A. Newcombe *et al.*, "Kinectfusion: Real-time dense surface mapping and tracking," in *IEEE ISMAR*. IEEE, October 2011. Online: http://research.microsoft.com/apps/pubs/default.aspx?id=155378

[6] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. Online: http://doi.acm.org/10.1145/237170.237269

[7] Altera opencl optimization guide. Online: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[8] "Implementing fpga design with the opencl standard," White Paper, Altera, Nov. 2013. Online: http://www.altera.com/literature/wp/wp-01173-opencl.pdf