GPU accelerated Realtime 3D Modelling with Microsoft Kinect

Paul Caheny

August 24, 2012

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2012

Abstract

Realtime 3D modelling of the environment and objects in it using a freehand Microsoft Kinect has recently been demonstrated. The Simultaneous Localisation and Mapping (SLAM) technique employed was possible in realtime due to almost the entire computational pipeline for the system being executed by GPGPU. Such modelling systems are of wide interest in Computer Vision, Robotics and Augmented Reality research but also have more practical applications such as 3D scanning.

The Point Cloud Library (PCL) , is a wide ranging open source software project for 2D/3D image and point cloud processing which is developing a Kinect & GPU based 3D scanning system named KinFu. This MSc. dissertation has evaluated the compute performance of the existing KinFu implementation and investigated possible performance improvements both on its existing CUDA platform implementation and via prototype porting to OpenCL for access to a wider range of hardware vendors and architectures.

A 30% increase in the overall performance of the CUDA KinFu app was achieved by the removal of an unnecessary type conversion which is a particularly expensive operation on the Kepler generation of NVidia GPUs The most computationally expensive kernel of KinFu was successfully ported to OpenCL and tested on an AMD GPU.

Contents

List of Tables

List of Figures

Acknowledgements

Thanks to Dr. Michele Weiland of EPCC for supervising this dissertation and providing invaluable direction with the planning and focus of the work. Javid Khan of Holoxica Ltd. provided the inspiration to work on 3D modelling with the Kinect. I would also like to thank Dr. Alan Gray and Dr. David Henty of EPCC who provided advice related to dense linear algebra on GPUs.

Chapter 1

Introduction

1.1 Background

3D model construction is a long standing problem in the area of computer vision with particular pertinence to Robotics and Augmented Reality applications. Currently there is also growing interest in constructing 3D models of real world objects for more practical applications such as 3D printing and retail display in e-commerce. Holoxica Ltd. are a company based at the Scottish Microelectronic Center at King's Building Campus, University of Edinburgh who specialise in 3D display & holography research and the production of holograms based on existing technology for a wide range of customers. This often necessitates the construction of 3D models from real world objects and thus Holoxica are interested in the possibility of using the Microsoft Kinect to construct 3D models of real world objects.

The Microsoft Kinect sensor which was released in 2009 has been a disruptive product in the computer vision marketplace due to its incorporation of a depth sensor with an unprecedented quality to cost ratio. Structured light depth sensors such as the one in the Kinect produce depth maps by emitting a static infrared light pattern on their environment and using an infrared sensor to detect deformation of the light pattern as it encounters surfaces. The resulting depth maps can be used to reconstruct 3D models of the environment.

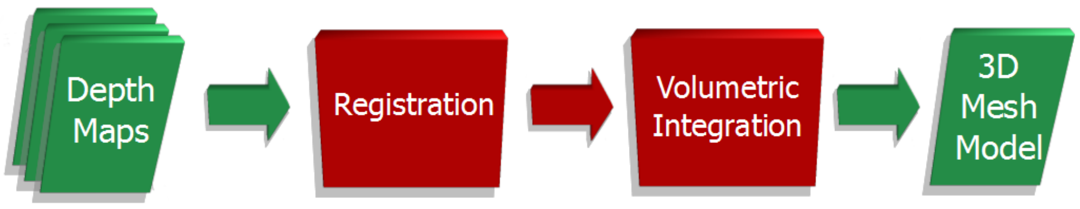


Figure 1.1: High Level Kinect Fusion Workflow

Although the depth sensor in Microsoft's Kinect was primarily designed for gesture control Microsoft Research recently demonstrated a system named Kinect Fusion [6] which used the data from the Kinect's depth sensor to construct realtime 3D models of medium sized scenes (e.g. a desk or a person sitting in a chair). The Kinect Fusion system employed a Simultaneous Localisation and Mapping (SLAM) technique allowing handheld and irregular motion of the sensor with automatic tracking of its pose plus synchronous realtime construction of the finished 3D model. This display of realtime performance for synchronous sensor tracking and model construction was a first and was enabled by a GPGPU accelerated implementation. Almost the entire computational pipeline of the system ran on a GPU.

In 3D model construction realtime performance is an important characteristic for many applications as it provides the operator with feedback about the quality of the finished 3D model during scanning of the object and thus allows the operator to direct the scanning in order to achieve the best results. For example the operator can focus the scanning on an area of the object which is incomplete in the model as displayed in realtime. Earlier systems which captured the raw data first and later construct a 3D model do not allow for this type of workflow and possibly necessitate repeated iterations of the scanning and model construction phases before a satisfactory end product is achieved.

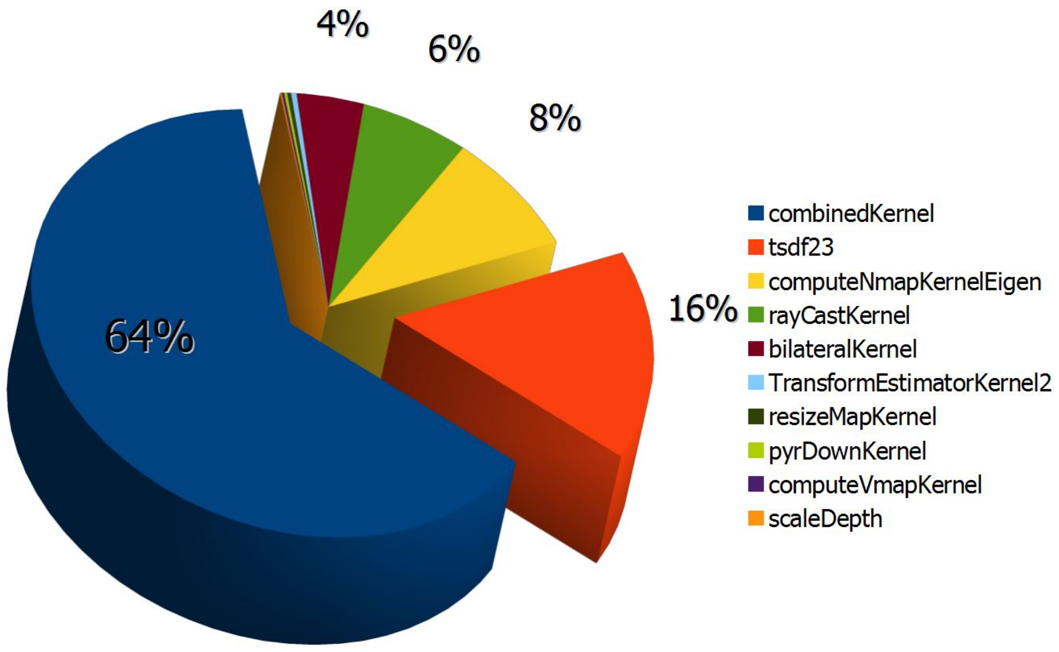


Figure 1.2: Proportion of computational time per GPU Kernel in the PCL KinFu implementation. The combinedKernel kernel performs registration and the tsdf23 kernel performs volumetric integration of the incoming depth data frame.

The Kinect Fusion system's computational pipeline is centred around 2 broad phases as seen in Figure 1.1, both primarily executed on the GPU. The first phase is to register the incoming depth maps from the Kinect by an algorithm named Iterative Closest Point (ICP), which provides the tracking of the pose of the sensor from one frame to the next and allows each incoming frame to be placed in a unified global coordinate space. The second stage employs a technique named Volumetric Integration which computes a signed distance function from each input depth map frame and accumulates the data into a discretised volumetric representation of the global coordinate space.

These two broad phases of Kinect Fusion's computational pipeline are the most conceptually complex and also the most computationally expensive as evidenced by the distribution of wallclock time per GPU kernel as measured by Nvidia's Nsight Profiler in Figure 1.2.

1.2 Changes to Dissertation Plan

During the early stages of the dissertation planning there was no publicly available implementation of a system based on Microsoft's Kinect Fusion. Furthermore while the Kinect Fusion system presented by Microsoft Research was targeted at Augmented Reality applications and medium to large size scenes using a desktop class GPU, Holoxica were interested in using the Kinect specifically for scanning real world objects of small to medium size. The original dissertation goal was to implement a system from scratch based on the ideas presented in Microsoft's Kinect Fusion research but tailored specifically to Holoxica's requirements. We also planned to investigate the possibility of running such a specifically tailored system on a laptop class GPU to enable scanning objects with a Kinect and laptop computer.

At the beginning of June we became aware of a freely available BSD licensed implementation of the Kinect Fusion system by the Point Cloud Library[22] (PCL) project. This software is (as of writing) still in a developmental stage and not officially released by PCL, however it is available to download and compile from PCL's development branch. With a freely available implementation of the Kinect Fusion system already available we reassessed the project plan and decided to base a new project plan around the PCL KinFu implementation rather than creating our own implementation of Kinect Fusion from scratch.

1.3 Motivation

The changes to the project plan as discussed above led to new motivation and goals for the dissertation. The existing beta version of PCL's KinFu application is implemented in CUDA, tying the system to Nvidia only hardware and precluding the use of the system on a wider variety of GPUs such as those from AMD and also a wider variety of heterogeneous processors such as desktop or laptop CPUs with integrated graphics hardware (e.g. AMD's Fusion APUs or Intel's new Ivy Bridge CPUs which support OpenCL access to the on-chip GPU) and mobile device SoCs.

PCL's KinFu system performs in a robust realtime fashion using the latest Nvidia desktop class GPUs such as Fermi and Kepler, however using laptop class GPUs its performance (while still usable) is below realtime and not capable of utilising the full 30 frames per second of data provided by the Kinect. This result in a lower quality experience for the user and ultimately a longer, less robust process to scan real world objects. Therefore outside of desktop class GPUs a faster implementation achieving closer to realtime performance is desirable.

The type of consumer depth sensing device pioneered by Microsoft Kinect is expected to be improved upon in future iterations of the product and in a proliferation of competing products. Thus, for example with much higher resolution input data from a future Kinect, better performing implementations may be required to achieve realtime performance even on future desktop GPU hardware. Kinect type sensors are also likely to appear in mobile devices in the near to medium term presenting new hardware architectures and performance limitations to the existing KinFu implementation.

In light of the above we decided to focus the dissertation's work on:

- Opportunities for optimising the existing KinFu CUDA implementation.
- Prototyping an OpenCL port of the KinFu application to
  - Broaden the range of target hardware architectures to include the heterogeneous processor types already described.
  - Investigate the possibility of gaining better performance from traditional discrete GPU architecture processors from a different vendor, namely AMD.

1.4 Related Work

This dissertation is primarily related to the work from Microsoft Research on Kinect Fusion and the PCL's BSD licensed implementation named KinFu which stems from it. Kinect Fusion builds on a number of techniques in computer vision and 3D modelling which have been refined over the past three decades. The main achievement of Kinect Fusion was to combine these techniques in a novel way utilising the particular characteristics of the Kinect sensor's output and leverage a GPU implementation to perform dense realtime SLAM. The published Microsoft Research did not make available any code or binaries for the Kinect Fusion system and contained some information about the performance characteristics of the overall system as a a whole.

PCL's KinFu application has not yet been officially released as part of the Point Cloud Library but is freely available to download and compile from PCL's development branch. PCL are currently working on extending the basic functionality of KinFu to handle reconstructing larger 3D scenes or spaces by increasing the 3D volume KinFu is able to operate on, a characteristic which has so far been limited by the amount of memory available on a single GPU. The KinFu tool as implemented by PCL is heavily dependent on pre-existing lower level functionality in the PCL library for handling visualisation, I/O and interfaces to sensor devices. It also uses functionality from the PCL library's external dependencies to other open source software projects. These include the Eigen library for CPU based linear algebra, the VTK library for visualisation and the OpenNI middleware and device drivers for interfacing with the Kinect sensor. PCL's KinFu implementation is CUDA only meaning the system is tied to CUDA enabled discrete graphics cards. As the KinFu system is unreleased and still in a development phase there is little in the way of detailed analysis of the performance characteristics of the various individual elements of the computational pipeline even on Nvidia hardware. The possible performance characteristics of the algorithms involved on other vendor's discrete GPUs or indeed other types of architectures such as heterogeneous processors is unknown.

1.5 Outline

Chapter 2 provides an introduction to 3D modelling from depth sensor data. In particular the techniques used in the Kinect Fusion and KinFu systems are presented with reference to how they work in isolation and how Kinect Fusion combines them into a novel pipeline. Chapter 3 presents an introduction to massively data parallel processors by describing Nvidia and AMD's latest generation discrete GPU architectures and Intel's latest generation Ivy Bridge processor which combines a traditional CPU with an integrated on chip GPU. All three products from these major manufacturers were release to the public in just the last few months and all feature new designs heavily influenced by the goal of providing or improving accessibility for general purpose data parallel computing on what was traditionally thought of as graphics specific hardware. Chapter 4 introduces the PCL library's implementation of the system described by the Microsoft Kinect Fusion research with a focus on the most computationally expensive elements of the implementation. Chapter 5 provides details of the work undertaken in profiling and optimising the existing KinFu implementation and porting elements of it to OpenCL prototypes. Chapter 6 presents the results of the work described in the previous chapter and the relevant performance analysis . Chapter 7 discusses the practical problems encountered which are inherent in designing application software for the disparate types of GPGPU devices and programming environments currently vying for supremacy in the heterogeneous HPC ecosystem. Finally Chapter 8 presents in summary the conclusions of this dissertation.

Chapter 2

Background to realtime SLAM with Kinect

2.1 Introduction

The Kinect Fusion system consists of a sequential pipeline of high level tasks to be performed on each incoming depth map from the Kinect. The two most complex elements of the pipeline are the registration of the incoming depth map frames and the volumetric integration of the data into a regular grid representation of real 3D space in the GPU's memory. Kinect Fusion's GPU implementation of these stages (including using a ray-casted model of the volumetric data as a reference to perform registration against) is also its most novel feature. The other computational steps represented by the kernels visible in Figure 1.2 can be thought of as preprocessing steps on the raw data before registration and post-processing the voxel grid to produce a human readable representation of the 3D model as an image or exportable 3D mesh. The rest of this chapter will describe the processing pipeline in detail, decomposed into pre-ICP registration, ICP registration, volumetric integration and post-volumetric integration stages. All stages and operations are carried out on the GPU unless stated otherwise.

2.2 Pre-processing before ICP

The Kinect's depth sensor produces 640x480 resolution depth maps at 30 frames per second (FPS). Unlike many other techniques for realtime 3D model construction the Kinect Fusion system does not rely on identifying features (for example geometric planes, corners etc.) in the incoming data in order to match sparse features between frames for registration. Kinect Fusion performs registration based on dense correspondences, i.e. correspondences from all data points in each depth frame. This dense correspondence matching and registration is enabled by the data parallel computational throughput of the GPU. As the incoming depth data from the Kinect Sensor is quite noisy the first step taken is to apply a bilateral filter to the depth frame which greatly reduces the noise present in the data while maintaining depth discontinuities. This is equivalent to the effect of bilateral filters in image processing which are used to remove noise and smooth an image while preserving edges.

After the application of the bilateral filter the depth data is replicated twice, halving the resolution each time by subsampling the data. This produces 3 views on a single depth data frame at native, 1/2 and 1/4 the Kinect sensor's resolution. All three views on the data are stored in a depth map pyramid. From here each level of the depth data pyramid is converted into a 3D point cloud from the point of view of the sensor by back projecting the depth data through the intrinsic image plane calibration parameters of the sensor. This point cloud representation of the data is stored as a vertex map where each vertex has 3 dimensional coordinates in the local coordinate space of the depth sensor. From this vertex map pyramid a normal map pyramid is computed based on a nearest neighbour calculation on the vertex map to give each vertex a value for the direction it is facing along with its position.

The vertex and normal pyramids are then stored for processing by the next step in the pipeline which is the ICP registration algorithm.

2.3 ICP Registration

In the following description the term target is used to describe the scene or object which is to be scanned with the Kinect for construction into a 3D model. The size of the target in KinFu is constrained by the resolving accuracy of the sensor at the lower end and by the amount of memory required to represent a given volume at a given resolution at the upper end. Reasonable limits are $50cm^3$ up to $3m^3$.

Another pertinent concept is the representation of all possible movement of a rigid body in three dimensional space as a combination of six degrees of freedom:

- Translating the rigid body along the x, y or z axes by a specific distance. This is equivalent to simply placing the object in a different position in space without rotating it. Each of these three degrees of freedom is represented by a single value for distance along the respective axis or more compactly as a single 3 element vector t for all three translations.
- Rotating the rigid body about its x, y or z axes. This is analogous to the aviation terminology of pitch (rotation which points the nose up or down), roll (rotation which points either wing up or down) and yaw (rotation which changes the heading or direction the nose is pointing along the ground). Each of these three rotational degrees of freedom are represented by a three element rotation vector or more compactly as a single 3x3 matrix R for all three rotations.

Thus a rigid body transform in three dimensional space is represented as the combination Rt.

The baseline use case for the Kinect Fusion system is that the data it gathers is from an environment where the target it is sensing is self-static, i.e. the sensor can move in relation to the target and/or the target can move in relation to the sensor but there can be no movement within the target in relation to itself. For example:

- When scanning a stationary desk full of items with a freehand Kinect, items on the desk cannot be moved during the scanning.
- When scanning a moving object or scene with a freehand Kinect, for example an item on a rotating turntable, the baseline use case is satisfied if the stationary background is disregarded (trivially achieved by disregarding all depth readings over a given distance threshold). Otherwise the baseline use case is not satisfied because the target is not self-static - the item on the turtable is moving in relation to the included background.

If the self-static target requirement is met both the target and the Kinect are free to move in six degrees of freedom. The relation between the sensor and the target from one frame to the next is simply the sum of the movement of the target with the movement of the sensor. For example after satisfying the self-static baseline requirement that the background is disregarded, rotating a target on a turntable through 360 degrees in front of a stationary Kinect is indistinguishable from the Kinect travelling in a 360 degree arc around the stationary object or any combination of the two.

In order for the system to track the relative pose of the target and sensor it computes a 3D rigid body transformation by the ICP process for the incoming depth map using the previous frame's pose as a reference. This allows the system to translate the incoming depth frames from the local coordinate space of the sensor into a fixed global coordinate space.

The global pose T_g of the sensor in relation to the target at time k is represented as:

$$T_{g,k} = [R_{g,k}, t_{g,k}] \tag{2.1}$$

where t is a 3 element vector representing the the three translating degrees of freedom along the x, y and z axes and R is a 3x3 matrix representing the three rotational degrees of freedom about the x, y and z axes.

The problem ICP solves then is given the current global pose, $T_{g,k}$ and a new incoming depth map, to compute the new global pose $T_{g,k+1}$. It does this by computing a transform:

$$T_{inc} = [R_{inc}, t_{inc}] \tag{2.2}$$

which registers the incoming depth map with the depth data as seen at the previous frame's pose, thus giving the change in pose from the previous frame to the incoming frame. The new global camera pose is then simply a composition of (2.1) and (2.2).

Given two identical rigid body surfaces separated in 3 dimensionsal space it is mathematically trivial to compute the transform that registers the bodies if correspondences between points on the two bodies are known. Correspondences in this case means points in successive depth frames which correspond to the same point in the real world target.

ICP uses this fact to make an iterative approximation to the exact transform by making a heuristic guess about which points correspond from one depth frame to the next. Given an initial guess about the pose of the incoming depth frame ICP guesses the point correspondences for all points in the incoming frame against the reference frame and calculates the transform which would register the two frames based on the initial guess for the pose and the guessed correspondences. If the initial guess for the pose was close enough this process will yield two more closely registered frames. ICP then reselects the point correspondences and computes the transform again, iterating this process until the two frames are aligned within some acceptable margin of error.

An important and novel characteristic of the Kinect Fusion system relates to a subtle requirement in the description above. ICP needs an initial baseline guess about the global pose of the the incoming depth map to start its iterative sequence before it selects any correspondences. Because the Kinect produces depth frames at 30 FPS and the GPU implementation allows for processing all frames in realtime without dropping any the change in pose from one frame to the next is likely to be quite small in natural handheld use of the Kinect to scan objects.

This allows for the initial guess about the transform for the incoming frame to be simply set to the actual pose calculated for the previous frame with a high degree of probability that it will be close enough for ICP to succeed in iteratively calculating the incoming frame's global pose without getting stuck in local minima. Therefore the high speed performance of the algorithm in processing the depth map data at the full frame rate of the Kinect does not just yield a higher quality resulting model, it also makes the system more robust in terms of tracking the relative motion of the sensor and target.

Given the initial guess for the global pose of the incoming frame (set to the global pose for the previous frame), ICP proceeds to find a corresponding point in the reference frame for each point in the incoming frame. It does this by projective association, sometimes referred to as reverse calibration. This technique transforms each point in the incoming depth data frame from the sensor's local coordinate space into the global coordinate space via the estimated global pose for the incoming depth data frame. The algorithm then takes this vertex in global coordinate space and works backwards from it into the local co-ordinate space of the previous depth data frame. It finds the vertex as seen by the sensor in the previous frame along the same ray from the sensor (i.e. recorded at the same x, y position in the depth data) and sets this vertex in the global space as the corresponding vertex.

The algorithm then computes the euclidean distance between the two points in global coordinate space and also the normal values at the two vertices. If the distance between them or the change in normal value exceeds a threshold value the correspondence is rejected as an outlier and does not form part of the computation of the transform required to align the surfaces.

This process results in a set of corresponding vertex pairs within global coordinate space. Each vertex also has an associated normal. In order to compute the transform which best aligns the surfaces given these correspondences a metric for the misregistration between the correspondences is posed as an optimisation problem. The metric employed by Kinect Fusion involves calculating the distance from the source vertex to the planer surface tangent to the destination vertex (rather than directly to the destination vertex itself). This allows the corresponding vertices to 'slide' along the planer surface tangent to the destination vertex and has been shown to result in faster convergence of the ICP process. The point to plane error metric is computed using the vertex pairs and their normals and then summed to give an L2 norm of the energy of the point to plane error metric across the entire depth map.

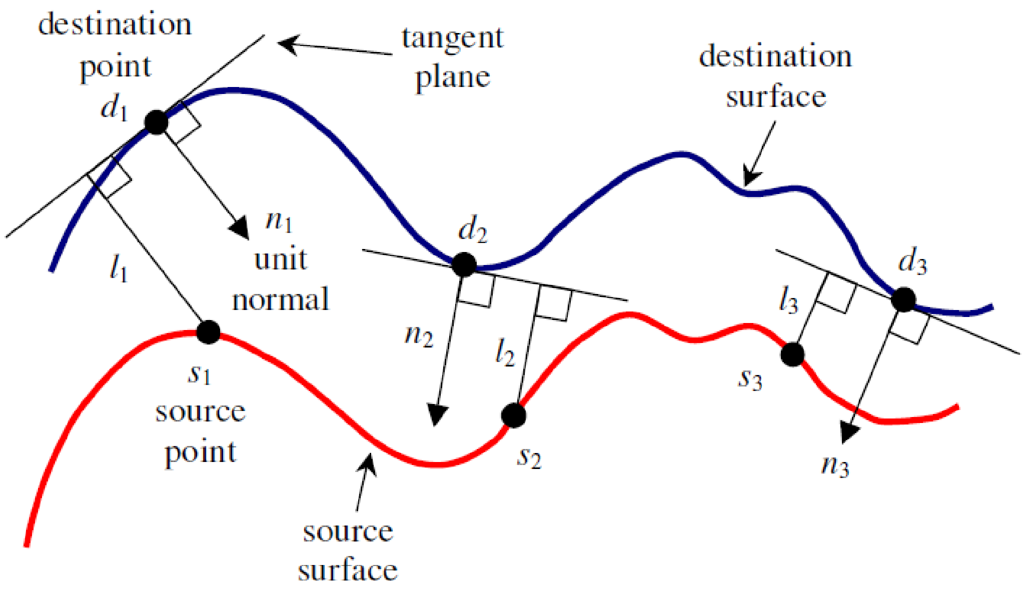


Figure 2.1: Point to Plane Error Metric, reproduced from [7]

The optimisation problem which minimises this point to plane error metric can be approximated by a 6x6 linear system based on the summation of the point to plane error metric for all the corresponding vertices. In the only part of the computational pipeline performed on the CPU, Kinect Fusion transfers the single 6x6 linear system to the CPU for solution by Cholesky Decomposition once per iteration of ICP. Kinect Fusion performs 4, 5 and 10 iterations respectively in a coarse to fine path through the three levels of the vertex and normal pyramids for the incoming depth frame.

2.4 Volumetric Integration

As a result of the ICP process the incoming depth frame can be placed in a unified global coordinate space. The process of volumetric integration discretises this global coordinate space into a regular 3D grid of voxels in the GPU's memory. For each incoming depth map the entire voxel grid is swept through and updated based on a signed distance function.

A signed distance function computes the relative distance value from a given point in 3D space to the surface in 3D space as captured by the sensor. For a given frame of depth data the individual vertices and normals are transformed into the global 3D coordinate space via the transform computed in the ICP stage. For each voxel in the 3D volumetric grid which is intersected by the ray running from the sensor plane through each captured vertex the relative distance from the voxel centre to the vertex is computed. This can be done for voxels both between the sensor plane and the vertex (i.e. outside the surface) and voxels behind the vertex (i.e. behind the surface) by following the ray through the surface.

For each vertex the absolute distance from the vertex to the sensor plane for the current depth frame and global sensor pose is computed. For each voxel intersected by the ray passing through the sensor plane and the vertex the absolute distance between the voxel centre and the sensor plane is computed. The difference between these two distances is the relative value of the signed distance function at the particular point in the global coordinate space represented by that particular voxel centre. This results in a positive distance value for voxels that lie between the sensor and a surface, a zero value if the voxel is intersected by a surface (i.e. a vertex lies within the voxel or the voxel is cut by a surface joining the nearest vertices) and a negative value if the voxel lies behind a surface as measured by the sensor.

As frames are continually added to the volumetric representation in memory each voxel stores an accumulated average of all the updates it receives by the integration of the signed distance functions for each depth map. This provides an average for the distance from that voxel centre to a surface, computed over all the times a ray passing through the voxel to a vertex was captured by the sensor. It is this accumulation and averaging of distance measures from the high frame rate of the Kinect that produces a high quality representation of the environment and ultimately an extremely high quality 3D model considering the relatively low resolution and noisy nature of the raw depth data in each individual frame from the Kinect sensor.

To further improve the end result as the voxels receive updated data for an incoming depth frame the incoming value for the signed distance function is weighted for incorporation into the voxel's average based on heuristics about the characteristics of the sensor and its accuracy. For example the accuracy of the Kinect's depth sensor technology is known to be better the more perpendicular the surface which the data point lies on is to the sensor's image plane. This means data points which the sensor captures from oblique surfaces are less reliable and accurate. A heuristic for weighting the data points based on this knowledge is achieved by using the normal map already computed in the preprocessing stage. The normal map gives each vertex a normal direction based on a nearest neighbour computation on the position of neighbouring vertices. The more normal (i.e. perpendicular) a surface is to the sensor's image plane at the vertex the higher the weighting the discretised values for the signed distance function in the voxels along the ray which captured that vertex.

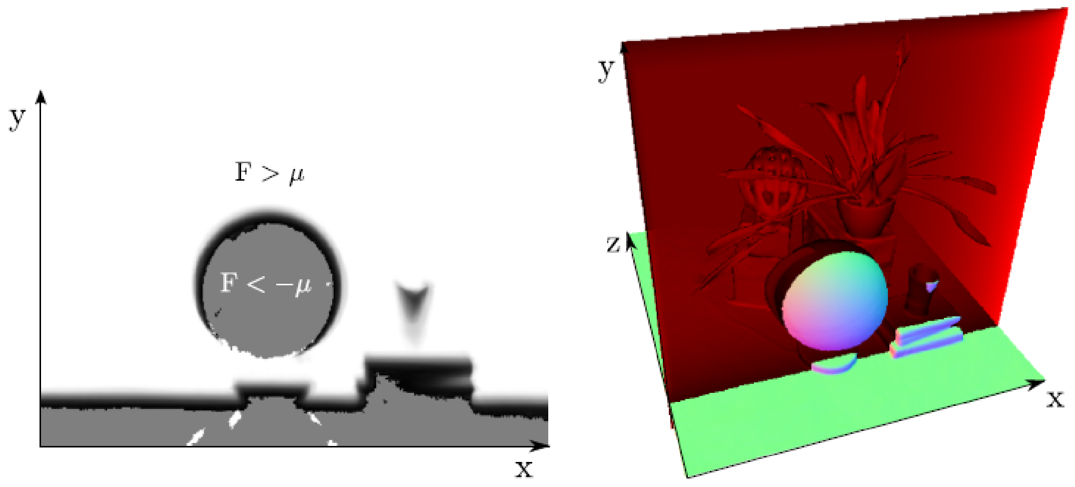


Figure 2.2: A view of a slice through the volumetric representation of the truncated SDF function. Grey areas are voxels which haven't been measured. White areas are not near a surface and there is a smooth gradient from white towards black (the zero-crossing) at surfaces, reproduced from [6]

In order to construct a surface representation or mesh based on the volumetric representation of the data it is not necessary to write discretised values for the signed distance function into all the voxels along the ray which captured the vertex. By making an assumption that the raw depth data from the sensor is accurate to within +/- a given margin of error, in practice only voxels within a threshold distance (related to this margin of error) from a surface in front of and behind the vertex are updated. Voxels in the 3D volume which are not near any surface as captured by the sensor are ignored.

2.5 Post-processing after Volumetric Integration

In order to visualise the volumetric representation of the data as stored in GPU memory a raycasting technique is used. In the case of Kinect fusion however this stage in the pipeline is leveraged to provide more than simply visualisation of the ongoing model construction to the user.

The raycasting technique involves traversing the individual rays from the global sensor pose through the voxel grid until a zero-crossing (value changing from positive to negative) of the discretised truncated SDF is observed. If no such zero crossing is observed during traversal of the voxel grid for an individual ray then no surface exists within the volume on that ray. If a zero crossing is found while traversing the ray the gradient of the truncated SDF can be computed by interpolating the discrete SDF values over multiple voxels around the zero crossing. The normal at the surface represented by the zero crossing in the SDF is then orthogonal to the gradient in the SDF. This normal for the surface at that particular point allows the raycasted image to be lit or shaded appropriately for visualisation but is also crucial for the second use of the raycasted model described next which is novel to Microsoft's Kinect Fusion system.

This other more important use of the raycasting technique is in the ICP phase. ICP as already detailed registers each incoming frame of depth data from the sensor against data obtained at the sensor pose from the previous frame. While it would be possible to simply register the incoming raw depth data frame from the Kinect against the previous raw depth data frame this would result in loss of tracking of the sensor over a large number of frames (as generated by the full 30 FPS provided by the Kinect). This would occur due to small but systematic error introduced by the pose estimation in each individual frame accumulating in the global pose estimate over time.

As the ICP process produces an estimated transform by an iterative process rather than an exact solution for the frame to frame pose estimation it by definition introduces some error to the global pose estimate in each successive frame. Using the raw depth data from the previous frame as the reference for the incoming frame would also negatively impact the accuracy of ICP due to the quality of the raw depth data frames which the low cost depth sensor in the Kinect produces. The individual frames include many points where no data was retrieved due to the physical characteristics of the depth sensor meaning less valid correspondences between frames for the ICP process to work with.

In order to avoid these problems Kinect Fusion employs a novel design which closes the loop in the 3D reconstruction pipeline already described. A raycasted image of the global volumetric data set is computed using the global sensor pose of the previous frame and used as input to the ICP phase for the next raw depth data frame from the Kinect.

ICP uses this high quality raycasted image of the volumetric representation of all the accumulated data to construct a virtual vertex and normal map taken from the point of view of the previous frame's global pose. It uses this virtual vertex and normal map instead of the vertex and normal map as computed from the previous raw depth frame and uses this as the reference against which to perform ICP for the incoming frame.

Microsoft's Kinect Fusion research has demonstrated the efficacy of, and indeed requirement for, this technique in maintaining accurate tracking of the global sensor pose with nosiy, high frame rate data as produced by the Kinect sensor. The raycasted image of the 3D model computed for each frame can also be displayed to the user to give a live video 3D visualisation of the model as it undergoes construction from the viewpoint of the sensor. It is also trivial to view the model by the same raycasting process in realtime from a user controlled virtual camera instead of the viewpoint of the sensor.

In a final post-processing step at a time of the user's choosing a file based representation of the model can be obtained by running the marching cubes algorithm on the voxel grid to extract a mesh based representation of the volumetric data.

Chapter 3

Latest generation GPU architecture

GPGPU programming has matured in recent years to the stage where the hardware and the techniques for using it for general purpose computing stand on their own independent of the purpose these devices were initially developed for, namely accelerating computer graphics.

The benefits of using this style of hardware for general purpose computing is now firmly established and this has led to a change in the architecture of everything from processors for handheld mobile devices to nodes within the most powerful supercomputers in the world.

Mobile devices have long pursued a design where massively parallel accelerators have been included in a SoC design with a more traditional CPU in order to minimise the system's power envelope.

Intel and AMD are now both incorporating massively parallel GPU style processors onto the same chip as their traditional CPUs and plan to devote a higher proportion of die space to such accelerators in future product lines.

Intel and AMD are both pursuing aggressive design targets in the discrete GPU market. To this end both manufacturers this year released the latest generation of GPU hardware with a strong influence in their respective designs having come from ther requirements of non-graphics computing workloads.

3.1 Nvidia's Kepler GPU architecture

Nvidia's new Kepler GPU architecture debuted this year in the consumer space with the release of the GeForce GTX680 (8 SMXs) and GTX670 (7SMXs) based on the GK104 chip and will be followed later this year by the Tesla K20 GK110 chip for professional applications. Both variants share the same Kepler architectural DNA. Both units are based on Nvidia's latest Streaming Multiprocessor X (SMX) an evolution of the SM found in the previous Fermi generation of chips. The consumer GTX680 Kepler already launched has 8 SMXs while the forthcoming Tesla K20 will have 15 SMXs.

Nvidia has significantly altered the design of the SMX in comparison to the SM in the previous Fermi generation. The most significant change is an increase in the number of CUDA cores per streaming unit. The Fermi generation SMs contained 32 or 48 CUDA cores while Kepler's SMX contains 192. The increase in CUDA core count per SMX is balanced by a slower clock rate for the CUDA cores. In Fermi the CUDA cores ran at an internal hot clock rate which was twice the speed of the GPU clock that controls the processor's backbone infrastructure. Kepler has removed the hot clock altogether so everything on the processor runs at the same GPU clock speed. The warp size which is Nvidia's smallest unit of concurrent SIMT execution remains at 32 threads. In Fermi a single warp of 32 threads executed a single instruction on 16 CUDA cores over two cycles of the hot clock which appeared to an application as a full warp instruction being executed over a single cycle of the GPU clock (which is half the speed of the hot clock). In Kepler with no hot clock a single warp instruction is executed over a single GPU clock instruction using twice as many CUDA cores, 32. Hence the increase in number of CUDA cores mitigates the removal of the hot clock transparently to the programmer. Figure 3.1 shows the new SMX design.

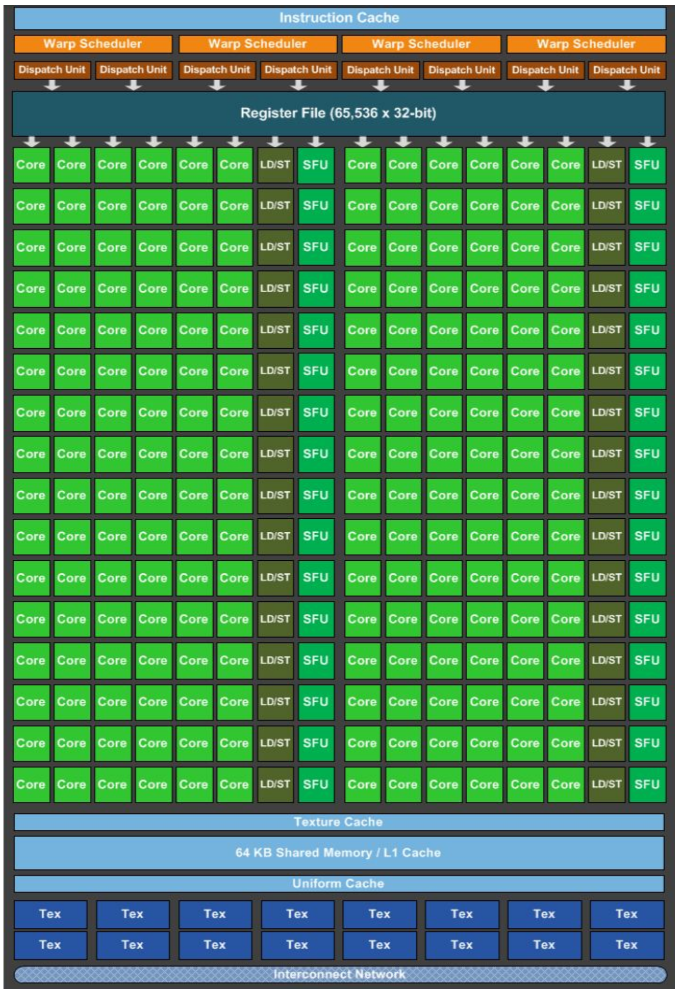


Figure 3.1: Nvidia's Kepler architecture SMX, reproduced from [11]

The supporting infrastructure for the CUDA cores in the SMX has also been redesigned. The register file doubles in size from 128KB to 256KB and the number of Warp Schedulers is doubled from two to four. The 64KB of local SMX memory which is application configurable as either 16KB/48KB or 48KB/16KB as Shared Memory/L1 Cache Figure stays the same from Fermi's SM.

3.2 AMD's Graphics Core Next architecture

AMD new GPU architecture named Graphics Core Next (GCN) is a radical departure from their former products. In AMD's previous generation GPUs the architecture relied on a Very Long Instruction Word (VLIW) approach to extracting instruction level parallelism (ILP) scheduled statically at compile time rather than dynamic hardware scheduling at runtime. This architecture was successful for graphics processing however it proved too inflexible for more general purpose computing on the GPU where runtime data dependencies resulted in less than optimal utilisation of the arithmetic logic units.

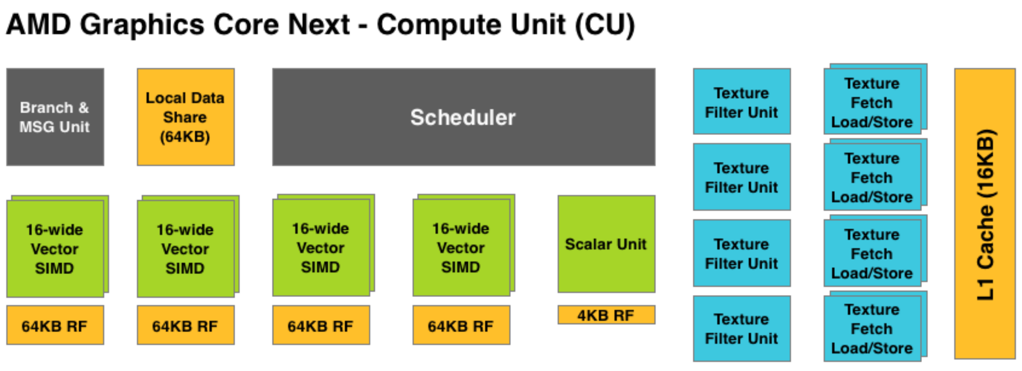


Figure 3.2: AMD's GCN architecture CU, reproduced from [21]

.

This prompted AMD to pursue a more dynamically scheduled SIMD style architecture for its new generation of products based on GCN. A GCN processor is made up of Compute Units (CU) as opposed to SMXs in Nvidia's design. Each GCN CU contains four vector units, each in turn containing 16 Arithmetic Logic Units (ALUs). This gives a total of 64 ALUs per CU in GCN which are analogous to what Nvidia's design terms CUDA Cores. AMD's flagship processor utilising the new GCN architecture the Radeon HD 7970 was released this year in the consumer market with professional Firestream class cards due later in the year similar to Nvidia's staggered release of Kepler.

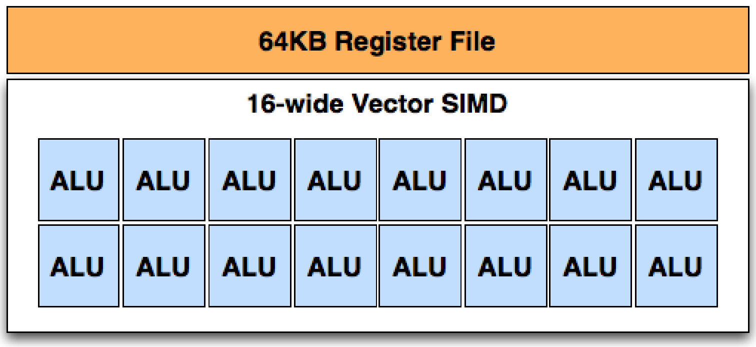


Figure 3.3: AMD's GCN SIMD unit, reproduced from [21]

.

The HD7970 is made up of 32 CUs, each containing 4 x 16 ALUs for a total ALU count of 2048 on the processor versus Nvidia's equivalent product the GTX680's 1536 CUDA Cores.

AMD’s design groups a 64KB register file with each vector unit for a total of 256KB of register file per CU, which is the same amount as Nvidia’s SMX has. Whereas Nvidia’s SMX has 64KB of local memory configurable as different splits of explicitly managed shared memory or hardware managed L1 cache, AMD’s GCN CU has distinct 64KB local shared memory and 16KB of dedicated L1 cache.

Chapter 4

KinFu's CUDA implementation

KinFu is the PCL library's open source implementation of the Kinect Fusion system as recently demonstrated by Microsoft Research. PCL was an existing library before KinFu development was undertaken and KinFu leverages some of the software which was already available within the PCL library, particularly for interfacing with Kinect-like devices and providing the raw depth data stream to the application. It also heavily relies on PCL's visualisation library to display the incoming raw data and the resulting 3D model graphically during model construction. This dissertation has focused on the compute specific parts of the application rather than parts of the application which perform ancilliary functions.

4.1 PCL Library Overview & Structure

The PCL library is a modern C++ library made up of a wide array of modular packages related to 2D and 3D image and point cloud data processing algorithms plus a number of higher level tools or application level packages which are built on top of the lower level packages. KinFu is one such application level package within the PCL library. While the KinFu application level package named kinfu_app handles all the setup and teardown of the application plus its interface to the Kinect depth sensor and graphical visualisation, all of the computational core of the application resides in a lower level library package named pcl_gpu_kinfu . pcl_gpu_kinfu contains all the computational kernels which make up almost entirely the runtime of the kinfu_app application. pcl_gpu_kinfu itself depends on one other low level PCL package named pcl_gpu_containers which implements templatised dynamic STLlike containers (with reference counted smart pointers etc.) as wrappers for raw pointers to GPU device side memory.

The PCL library as a whole has external dependencies to several open source software packages. Most pertinent of these to this work is the open source Eigen C++ template library for CPU based linear algebra.

4.2 PCL's KinFu application

PCL's kinfu_app package implements an application which manages the interface between the application and the Kinect sensor device and also the ongoing visualisation of the raw incoming depth data and 3D model construction during runtime. kinfu_app uses the OpenNI framework driver and middleware software in order to interface with the Kinect sensor and access the required raw depth data stream from the sensor. kinfu_app allows for running the system in a batch mode with prerecorded depth data in the OpenNI specified .rcd file format for depth data recordings. kinfu_app also allows for saving the resulting 3D model after a kinfu_app run by dumping the model to the open standard .ply 3D mesh format. After performing all setup and acquiring a handle to the OpenNI framework for access to the live or recorded depth data kinfu_app enters a task parallel multithreaded loop with one task each taking care of the incoming data, the visualisation and user interface and the interface to the computational backend of the application encapsulated by the lower level pcl_gpu_utils package. This design allows for running the computational backend of the application on a variety of different hardware with different performance characteristics. For example if the computational backend is executing on hardware which allows for processing depth frames in excess of the 30fps provided by the Kinect the application level design allows for the user interface and visualisations to run in excess of 30 FPS delivered from the Kinect sensor. Conversely if the computational backend is running on hardware which is not capable of handling incoming data at the full 30 FPS frame rate of the Kinect sensor then kinfu_app simply drops incoming frames from the Kinect and whenever the computational backend is ready to process a new frame provides the most recent one from the incoming data stream for processing.

4.3 KinFu's ICP Stage

PCL's ICP stage is implemented by a controlling loop (Listing 4.1) in the host code to describe the iterative process by which ICP arrives at its result. ICP performs a constant number of iterations per frame in order to keep the computational cost of the ICP stage constant in the system's pipeline. The iterative process works through the normal and vertex map pyramids produced by the preprocessing stage in a coarse to fine order, allowing the early iterations to be completed more quickly and moving to more expensive iterations on the full resolution data later as the transform becomes more accurate with each iteration. The vertex and normal map pyramids for the previous frame in global coordinate space and the current frame in the local coordinate space of the sensor are 3 levels deep at 1/4, 1/2 and native resolution of the sensor respectively. The number of iterations of ICP performed per level is 4, 5 and 10 as per Microsoft's Kinect Fusion system for a total of 19 iterations of ICP per incoming depth data frame.

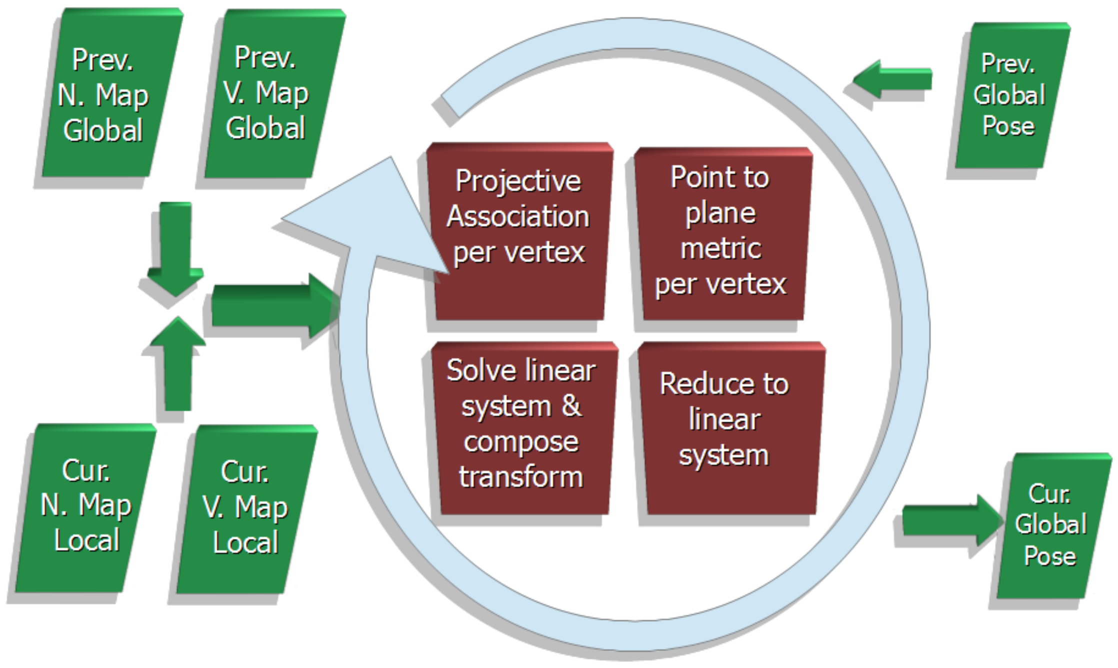


Figure 4.1: ICP Process. Previous and current frame normal and vertex map pyramids as input plus previous global sensor pose. Current frame global sensor pose as output of iterative process.

Each iteration of the ICP loop comprises four sequential tasks:

- Matching corresponding points between the vertices in the current vertex map and the previous vertex map by projective association.
- Computing the point to plane error metric between the corresponding vertices.
- Reducing the frame wide point to plane error values to a single linear system consisting of a 6x6 symmetric positive definite matrix A and 6 element solution vector b.
- Solving the resulting single Ax=b linear system and accumulating the solution into the current estimate for the current frame's global pose.

At a high level KinFu decomposes these four tasks into three CUDA Kernels to be executed on the GPU and an invocation of the Eigen library's linear algebra solver on the CPU.

Listing 4.1: ICP Host side Pseudocode

```
/* for each level in the vertex & normal map pyramid */
for (int level_index=LEVELS-1; level_index>=0; --level_index) {

  /* ...get vmaps & nmaps from current & previous pyramids...*/
```

```
    /* for 4, 5 or 10 iterations depending on pyramid level  */
    for (int iter = 0; iter < iter_num; ++iter) {

      /* ...setup combinedKernel... */

      /* Launch combined kernel to associate corresponding points
         and compute point to plane error metric per correspondence.
          */
      combinedKernel<<<grid, block>>>(cs);

      /* ...setup TransformEstimator2 kernel... */

      /* Launch TransformEstimator2 kernel to reduce error metric to
         linear system. */
      TransformEstimatorKernel2<<<TranformReduction::TOTAL,
         TranformReduction::CTA_SIZE>>>(tr);

      /* ...copy 27 double values making up symmetric 6x6 linear
         system elements A and b from GPU to CPU memory space...*/

      /* ...check linear system's determinant, reset tracking &
         entire system if matrix too ill-conditioned...*/

      /* ...solve linear system using Cholesky Decomposition by
         Eigen library...*/

      /* ...convert linear system solution x back to 3x3 3D rotation
          matrix R and 3 element translation vector t representing
         the 3D rigid body transformation for this ICP iteration...
         */

      /* Compose incremental Rt onto accumulated estimate for
         current frame. */
      tcurr = Rinc * tcurr + tinc;
      Rcurr = Rinc * Rcurr;
    }
  }
```

The full resolution data maps (current and previous vertex and normal maps) are 640 x 480 elements (with the next two levels in the pyramid halving the resolution again). Each element for both the vertex and normal maps contains three float values (x, y and z dimensions for vertex position and x, y and z magnitude for normal direction respectively). KinFu stores these maps as two dimensional arrays in global device memory in a structure of array (SoA) scheme (i.e. all the x values stored contiguously, then all the y values and finally the z values).

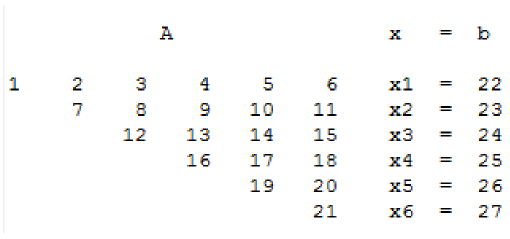
The first kernel launched on the GPU by the ICP loop is the combinedKernel. This kernel is responsible for two tasks, namely computing the correspondences between the vertices in the incoming data and the reference data based on the pose of the sensor at the previous frame and computing the point to plane error metric per correspondence.

For the first of these tasks the code is split into a separate __device__ kernel (a kernel callable from another kernel in contrast to a __global__ kernel which is callable only from the host). This kernel which performs the correspondence matching is described in Listing 4.2

Listing 4.2: Projective Association Kernel Pseudocode

```
__device__ __forceinline__ bool
search (int x, int y, float3& n, float3& d, float3& s) const
{
  /* Get float3 normal & vertex for incoming depth frame at
     position [x][y] from global memory current frame data maps */

  /* Trasform vertex to global coordinate space based on current
     global pose estimate Rt for frame */
  float3 vcurr_g = Rcurr * vcurr + tcurr;

  /* Work backwards from global coordinate space into local
     coordinate space of previous frame */
  float3 vcurr_cp = Rprev_inv * (vcurr_g - tprev);

  /* calculate x, y coordinates in the image plane of the sensor
     for the reverse calibrated vertex */

  /* Get float3 normal & vertex from previous frame for reverse
     calibrated x, y coordinates from global memory previous frame
      data maps */

  /* Correspondence rejected as outlier if euclidean distance
     between vertices too great */
  float dist = norm (vprev_g - vcurr_g);
  if (dist > distThres)
    return (false);

  /* Transform sensor coordinate space normal to global coordinate
      space based on current global pose rotation vector R */
  float3 ncurr_g = Rcurr * ncurr;

  /* Correspondence rejected as outlier if normal difference
     between corresponding vertices too great */
  float sine = norm (cross (ncurr_g, nprev_g));
  if (sine >= angleThres)
    return (false);

  /* Correspondence found & valid, return corresponding vertices
  and normal for corresponding vertex in previous frame */
  n = nprev_g;
  d = vprev_g;
  s = vcurr_g;
  return (true);
}
```

Once this kernel returns its result, namely the corresponding vertices in the global coordinate space the combinedKernel continues to calculate the point to plane error metric for the two corresponding points based on the corresponding vertex positions and their normals. As the resulting linear system approximation from the point to plane error metric consists of a 6x6 symmetric matrix A and 6 element vector b there are in total 27 unique values computed that make up the linear system. 21 values make up the upper triangle of A which are simply replicated with opposite sign in the lower triangle and six elements for b. In order to produce the complete values for the linear system which represents the entire set of correspondences the kernel must do 27 reductions, one for each resulting element of the linear system.



		A				x	=	b
1	2	3	4	5	6	x1	=	22
	7	8	9	10	11	x2	=	23
		12	13	14	15	x3	=	24
			16	17	18	x4	=	25
				19	20	x5	=	26
					21	x6	=	27

Figure 4.2: The 27 unique values which make up a 6x6 linear system Ax = b where A has symmetric values with opposite signs

For the full 640 x 480 resolution two dimensional data maps the combinedKernel is launched with a block dimension of x = 32, y = 8 threads (256 threads per block) and these blocks then make up a grid of dimensions x = 20, y = 60 (1200 blocks) to cover the entire 640 x 480 data maps.

The results of the combinedKernel kernel are stored in a two dimensional array for the 27 linear system elements per CUDA block in the combinedKernel kernel launch, i.e. a 27 x (20 * 60 = 1200) 2D array. The final linear system is simply a reduction of the 1200 per block values for each of the 27 linear system variables and this reduction is performed by the TransformEstimator2 kernel. The reductions in both combinedKernel and TransformEstimator2 kernel are implemented by templatised __device__ functions as per suggested best practice in Nvidia's programming guide which utilises warp synchronous techniques to perform the reduction optimally in GPU memory.

4.4 KinFu's Volumetric Integration Stage

Chapter 5

Optimising KinFu in CUDA & porting to OpenCL

5.1 Removing CPU involvement from the ICP loop

As can be seen from Figure 1.2 the combinedKernel kernel which is called from the main ICP loop described in Listing 4.1 is responsible for 64% of the computational wallclock time on the GPU. The measured runtime for the combinedKernel kernel includes the runtime for the `__device__` search kernel in Listing 4.2 as this kernel is called from the combinedKernel on the device and is thus inlined in combinedKernel by the CUDA toolchain. The other kernel called by the ICP loop, TransformEstimatorKernel2 is responsible for an insignificant fraction of the runtime. As already described in 2.3 the ICP loop runs for a total of 19 iterations over a coarse to fine path through the three levels in the vertex and normal map pyramids. For each of these iterations the 27 elements which make up the linear system approximation for the optimum registration transform for that iteration must be copied from the GPU to the CPU memory where, as can be seen in Listing 4.1, the Eigen library is used on the CPU to calculate the determinant of the matrix A in the linear system before solving the linear system. Eigen is also used to coerce the solution of the linear system back into a 3D rigid body transformation representing the incremental transformation for this iteration of ICP. This incremental transformation is then composed onto the running total for the estimated global pose transform before the loop ends and a new iteration of ICP begins.

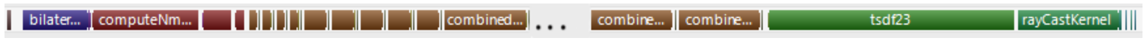


Figure 5.1: Timeline of execution of KinFu for a single frame of depth data (7 of 19 iterations of combinedKernel ommitted at centre

.

The timeline of events in the computational pipeline can be seen in Figure 5.1 which has truncated some iterations of combinedKernel for presentation.

The first two kernels with partial names in Figure 5.1 are the bilateralKernel and computeNmapEigenKernel which are part of the pre-processing stage in advance of ICP as described in 2.3. There are 12 out of 19 instances of combinedKernel shown in Figure 5.1, all are of the same colour. The first four invocations are of similar short duration corresponding to the 4 ICP iterations in the lowest level of the data map pyramids. There follows 5 longer iterations in the next level of the data map pyramids and finally the 10 (of which 3 are shown in Figure 5.1) iterations in the data maps corresponding to the native resolution data from the Kinect. As can be seen from the timeline there is a small break in computation on the GPU between each of the 19 invocations of combinedKernel. As per the ICP loop described in Listing 4.1 in each of the intervals between invocations of combinedKernel the transformEstimatorKernel2 runs to do a final reduction on the point to plane error data from all the CUDA blocks which participated in the combinedKernel. This reduction is followed by a CUDA Device to Host memory copy of the linear system for solution and composition into the global pose estimate on the CPU. This is followed by the CUDA setup and launch by the host of the next combinedKernel call.

Although the time taken to copy the data for the linear system to the CPU, solve the linear system with Eigen on the CPU and compose the result onto the global pose estimate between invocations of combinedKernel is small the first step taken with the code was a transformation to remove the GPU to CPU data copy and move all computation within the ICP loop to the GPU. This was a not a trivial task as GPUs are not well suited to solving dense linear system without any involvement of the CPU and there is no freely published software to do this to the authors knowledge. The most widely used library for GPU accelerated high level linear algebra computations is the MAGMA library. However even though the MAGMA library provides LAPACK style linear algebra solver routines with interfaces for data located in GPU memory in all cases MAGMA copies the data to the CPU's memory space before the CPU decomposes, packages and directs the solution of the linear system with the assistance of the GPU. MAGMA is designed for much larger linear systems than required in this case. After some research a suitable code was discovered on the registered Nvidia Developer website. This access protected area of the Nvidia website is used to distribute experimental software to registered CUDA Developers. The code which was recently published there by Nvidia solves small linear systems (up to matrices of about 60 rows and columns) in parallel entirely on the GPU. This code was incorporated into the KinFu application's ICP implementation to remove the dependency on the Eigen CPU solver. With the addition of some more naive single threaded CUDA code to compute the determinant of the 6x6 matrix on the GPU and coerce the linear system solution back into a 3D rigid body transform and compose the incremental transform on the global pose estimate all data copies from the GPU to the CPU were removed and all computations within the ICP loop in Listing 4.1 were being performed exclusively on the GPU.

Given the small amount of time taken for the deivce to host memory copy and CPU solution of the small linear system in the original code relative to the other kernels involved in the ICP loop a significant beneficial impact on the runtime for the application was not anticipated. However at the outset it was hoped removing the CPU involvement from the ICP loop might enable further transformations to the code on the device. For example possibly removing the need for the 19 separate kernel invocations and their associated overhead costs in setup and synchronisation or to allow for pipelining of the computation in multiple streams on the GPU.

5.2 Focusing on ICP

As we have already shown ICP is by a very significant margin the most computationally expensive stage in the KinFu pipeline, with the combinedKernel kernel responsible for four times the runtime cost of the next most expensive kernel as per Figure 1.2. It is also clear from a reading of the code that it is the most complex portion of the application from a conceptual perspective both on the host and device side. For this reason it was decided to focus the analysis and optimisation efforts in this area.

As previously described the pcl_gpu_kinfu package has a dependency towards another PCL library package named pcl_gpu_containers . This package provides host side applications with object oriented dynamic C++ conatiner style wrappers for raw pointers to areas of memory on the device. This functionality is extremely beneficial from the point of view of a CUDA application developer. However when analysing performance, trialing optimisations and prototype porting to OpenCL (which doesn't support many of the language features used by the pcl_gpu_containers framework) this functionality is a hindrance to effective and timely progress.

In order to make the performance analysis and search for optimisation opportunities more focused we decided to create a benchmarking harness code for the ICP stage in isolation. This would allow for more rapid analysis, experimentation and porting of the ICP implementation. The benchmarking harness code created is named KF_Harness . This harness code sets up all the memory and data resources required by the ICP loop but uses raw pointers and arrays instead of the functionality provided by the pcl_gpu_containers package. The harness invokes a single iteration of the exact ICP loop used in the full application. For this single iteration we used the full resolution data (of the three level data pyramid used during an entire ICP phase) as input. Performance of the ICP stage is extremely stable across frames (i.e. runtime is the same with different input data instances) in the full application so focusing optimisation on a single iteration and single set of data is representative.

We dumped the relevant input data to binary files from a regular run of the full KinFu application along with the resulting transform increment computed at the end of the ICP iteration. The harness reads this data from file and uses it as the input to the single ICP iteration. The single ICP iteration performed by the test harness with raw memory management was observed using the Nsight profiler to perform almost identically in runtime to the same iteration as perfomed by the full application with the pcl_gpu_containers based memory management.

In order to verify the result of the ICP iteration at the end of the test harness execution the result computed is compared to the result dumped to file by the full application for the same input data. This allows for verification of any transformations attempted with the ICP iteration inside the harness.

5.3 Porting KinFu's ICP to OpenCL

5.4 CUDA versus OpenCL

As OpenCL is an industry wide standard designed to target not only a variety of vendors' hardware but a variety of architectures it lags behind CUDA in terms of language features and ease of use from the programming point of view. With the modern CUDA platform there are an increasing number of choices how to express an application. At the highest level (and a welcome development for non-programming professionals) is the OpenACC standard which provides a directives based approach to annotating existing sequential host code in a similar paradigm to OpenMP. The OpenACC enabled compiler will then entirely manage the movement of data and computation of the annotated parts of the code to the GPU without any further transformations to the code.

Currently there is also much interest around high level object oriented libraries for CUDA development such as Thrust, CUSP, MAGMA etc. These libraries provide modern object oriented C++ style interfaces to the programmer allowing more control over how the programmer expresses their algorithm than OpenACC for example. Such libraries abstract away almost all the details of the separate host and device memory spaces and allow the programmer to concentrate on expressing the what rather than how for their problem while at the same time leveraging highly optimised expert implementations specifically designed for GPUs behind the programmer facing API. These libraries represent an extremely attractive option for new greenfield application development to take advantage of the CUDA platform and insulate the application developer from the changes in hardware architecture from one generation of device to the next while at the same time offering 'free' performance boosts to applications when the libraries or underlying hardware progresses.

It is a sign of the rapid maturation of GPGPU computing that we can at this stage talk about 'legacy' CUDA code which was written before the availability of so many of the new features we can now take for granted. Before the availability of OpenACC and the type of high level libraries already mentioned programmers were forced to program separately for code to run on the device and the host and to explicitly manage execution and data across this interface. Indeed today this is still the level which offers the most flexibility to the programmer where the investment can be justified for the absolute highest performance.

To program separately for the host and device and manage that interface explicitly there are two levels of abstraction available on both the host and device side which are shown in table 5.1 and 5.2 respectively.

API Level	Abstraction Level	Style
CUDA Runtime	High	C
CUDA Driver	Low	C

Table 5.1: The host side APIs available for interfacing with the device.

The driver API provides a much higher granularity interface to the device with more explicit management of individual parameters and raw memory when launching and synchronising kernel execution on the device and when transferring kernel parameters or data between the host and device. It is also much more verbose which is why most applications use the runtime interface which for the vast majority of cases provides enough functionality to extract optimum performance from the underlying hardware.

Language	Abstraction Level	Description
CUDA C	High	C with some C++ features
PTX Intermediate Language	Low	Assembly-like

Table 5.2: The languages available for expressing code which runs on the device.

On the device side programming environment some of the important C++ like extensions to the C lanaguage which are provided by the CUDA C programming language include object oriented like grouping of data with methods, class inheritance, operator overloading for user defined classes, templatised classes and functions, functors (function objects) and references. Furthermore the Nvidia CUDA toolchain allows mixing the host and device side code in the same compilation units and automatically separates out the device and the host code at compile time.

The PTX intermediate language is an interface which is gaining increasing interest for applications in high performance computing. As a first introduction it can be very useful for examining in detail how the compiler is transforming the high level source code when performing detailed profiling of application performance. This can lead to hints or clues about performance bottlenecks in the source code which may not be readily evident from profiling tools alone. Furthermore PTX code can be written or edited by hand to give an assembler like programming interface for GPUs, with the CUDA toolchain supporting the compilation of PTX source code to binary format.

OpenCL by contrast supports only the bare essentials in terms of a host side API and device side programming environment. There are not at the moment any implementations which generate OpenCL code from directive based approaches such as OpenACC. Likewise both high level general purpose and scientific programming libraries for leveraging GPU acceleration are extremely limited compared to the plethora of CUDA based alternatives such as CUSP, Thrust, MAGMA, etc.

When comparing the options in OpenCL for low level host and device side application programming OpenCL also comes up very short in comparison to CUDA. On the host side OpenCL offers only an API at the level of the CUDA Driver API, an API which almost nobody ever chooses to program with on CUDA. This means translating CUDA host side code (typically written in the CUDA runtime API) to OpenCL host side code requires a huge bloat in code size and large decrease in code readability and maintainability, all for no performance benefit for the vast majority of applications.

On the device side OpenCL's programming language is a much more strictly C-like syntax, removing many useful features of modern programming languages which are largely taken for granted by application programmers these days.

Finally, the OpenCL toolchain is much less flexible than the CUDA one and requires explicit separation of host and device side code into separate compilation units. As OpenCL is designed to run on multiple target architectures it also employs a JIT compilation phase at runtime for the device side code making building, profiling and debugging of OpenCL code more cumbersome than with the statically compiled CUDA C device code.

5.5 Porting the KinFu benchmarking harness from CUDA to OpenCL

The PCL library in general including the KinFu application specifically make use of most of the modern language features provided by both C++ and the CUDA platform. This meant that on both the host and the device side a large development and testing effort was required to port from CUDA to OpenCL. On the host side the CUDA runtime API calls had to be translated to the lower level and more verbose OpenCL host side API.

On the device side the CUDA C kernel code had to be considerably refactored first to remove use of all the language features which OpenCL C does not support. This included:

- Splitting the host and device code into separate compilation units
- Refactoring device side code into separate data structs and standalone functions instead of the C++ style class encapsulation of data and functions supported by CUDA C
- Refactoring templatised device side kernels to static code device side kernels
- Refactoring to remove the use of functors
- Refactoring to remove the use of operator overloading for user defined types

After this refactoring was completed and verified the next step was to translate the relevant CUDA device side API calls to OpenCL. These include calls for synchronisation and obtaining access to the runtime environment for kernel configuration data (thread and block dimensions and IDs).

Chapter 6

Results and Performance Analysis

6.1 Removing CPU involvement from CUDA code

The removal of the CPU's involvement in the ICP loop is illustrated by Figure 6.1 and 6.3.

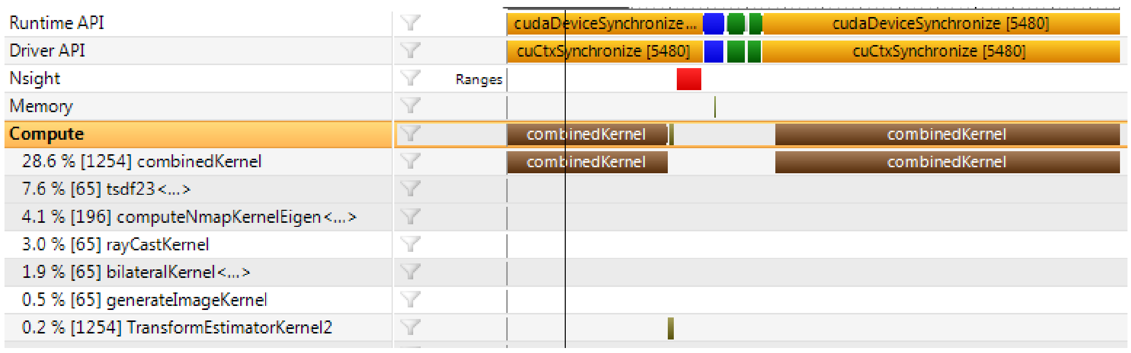


Figure 6.1: The Nsight timeline profile showing the stage between invocations of the combinedKernel in the original ICP loop

In Figure 6.1 we see the Runtime and Driver API on the host displayed at the top and the host/device memory interface followed by the individual device side kernels. As per the ICP loop described in 4.1 we see after the finish of an invocation of the combinedKernel kernel the TransformEstimatorKernel2 runs to do the final reduction of the point to plane error metric to the linear system. In the figure we then see in the Memory lane a very short memory transfer (which is from device to host) which places the linear system on the host. Although the host side computation of the solution to the linear system and the composition of the solution into the running total for the global pose estimate are not shown they are completing before we see the first two green coloured elements in the Runtime and Driver APIs at top which are the setup calls on the host for the next invocation of the combinedKernel kernel on the device.

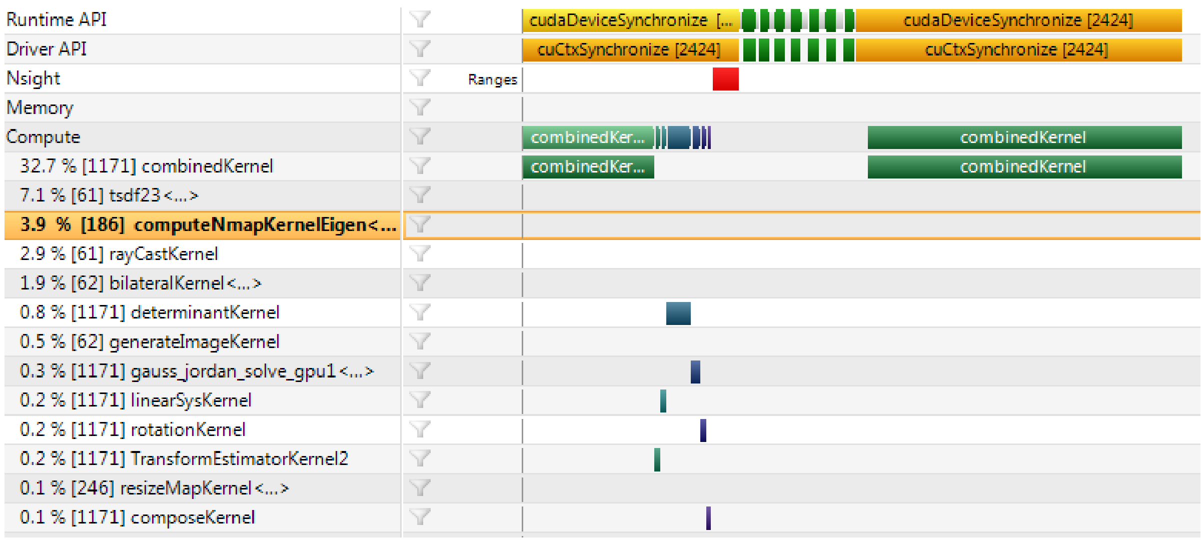

Runtime API
Driver API
Nsight
Memory
Compute
32.7 % [1171] combinedKernel
7.1 % [61] tsdf23<...>
3.9 % [186] computeNmapKernelEigen<...
2.9 % [61] rayCastKernel
1.9 % [62] bilateralKernel<...>
0.8 % [1171] determinantKernel
0.5 % [62] generateImageKernel
0.3 % [1171] gauss_jordan_solve_gpu1<...>
0.2 % [1171] linearSysKernel
0.2 % [1171] rotationKernel
0.2 % [1171] TransformEstimatorKernel2
0.1 % [246] resizeMapKernel<...>
0.1 % [1171] composeKernel
Ranges
cudaDeviceSynchronize [...
cudaDeviceSynchronize [2424]
cuCtxSynchronize [2424]
cuCtxSynchronize [2424]
combinedKer...
combinedKernel
combinedKer...
combinedKernel

Figure 6.2: The Nsight timeline profile showing the stage between invocations of the combinedKernel in the ICP loop after all computation was moved to the GPU

In Figure 6.3 we see that five extra kernels being run on the device after the TransformEstimatorKenrel2 kernel. These new device side kernels perform on the GPU all the functions which were being performed on the CPU in the original code. The GPU to CPU memory copy has been eradicated. At the top of the figure we can see extra green coloured CUDA API invocations on the host side compared to the original code. These are the setup and launch calls for the next combinedKernel and the extra kernels added to replace the CPU side code.

The time taken to perform the requisite computation between invocations of combinedKernel is greater than in the original code, which is not entirely surprising as most of the kernels added were naively implemented except for the gauss_jordan_solve_gpu1 which came from the code posted in the Nvidia CUDA Developer Zone as described earlier. There is also a time penalty involved with the setup and launch of such small, short running kernels.

It had been hoped at the outset that removing the CPU from the computational pipeline would present further opportunities for refactoring the code for greater performance by possibly rolling the 19 successive invocations to combinedKernel and the other requisite kernels into a single long running kernel launch or indeed by pipelining some of the computation into multiple streams. However at this stage it had become clear that neither possiblities were attractive. The multiple kernels require different launch configurations in terms of numbers of threads per block and number of blocks per grid and these parameters can only be specified with the launch of a new kernel and cannot be altered programmatically from within a launched kernel. Also the entire sequence of iterations of the ICP loop are sequential in nature with each step requiring completion of the previous step before continuing.

6.2 The discovery of missing performance in KinFu's CUDA implementation

As discussed in section 5.5 porting the code to OpenCL required a lot of refactoring of the device side code to remove features not supported by the OpenCL device side C99 based language. In the process of performing this refactoring a significant speedup in execution of the ICP loop as measured in the KF_Harness benchmarking code was observed. This increase in speed for the ICP loop in isolation was of the order of 40%.

In order to investigate this significant change in the code's performance detailed analysis and profiling was undertaken comparing the original CUDA code to the refactored CUDA code. The refactored code was verified as producing correct results by the method already described in section 5.5 and it did not seem plausible that any or indeed all of the more modern language features which were removed during the refactoring such as references, functors, user defined operator overloading or templatised functions could be responsible for such a significant change in performance.

The first step in analysing a performance of a CUDA code is to turn on the CUDA compiler's verbose output. This enables output from the compiler regarding the number of private registers used by each kernel. Private registers are maintained for each thread which makes up a kernel launch and are the fastest type of memory available to an executing thread, requiring zero extra clock cycles to access as part of an instruction in the absence of any read after write data hazards. As with any high performing resource registers are in limited supply. If a kernel requires more registers per thread than are available in the fast register file there are two options, both of which can have significant detrimental effects on performance. The first option is that the occupancy of the SMX can be lowered meaning less threads are resident, or 'in-flight' in the SMX at the same time, which allows more register space per thread. This can impact performance because a high number of in-flight threads in the SMX are required to enable the SMX to cover the latency of memory access by resident threads. When a thread is stalled waiting on an operand to come from L1 or L2 cache, shared memory or indeed global memory (400 - 600 clock cycles possibly) the SMX needs to swap other threads which are ready to execute onto the arithmetic logic, this prevents unused clock cycles on the arithmetic logic and increases achieved FLOPs. So, if the number of resident threads (which are grouped into warps of 32 threads, the smallest scheduling block used in NVidia's GPU architecture) in the SMX is reduced because of register pressure this reduces the SMX's ability to cover memory access latency, lowering achieved FLOPs.

The other option instead of lowering the occupancy of the SMX is to 'spill' the extra register storage required by the resident threads to what Nvidia terms local memory. This is somewhat of a misnomer as local memory is not embodied by any distinct physical memory on the GPU like shared, global and L1 or L2 cache are. Local memory in Nvidia's terminology is simply a thread private portion of global memory reserved when required due to register spilling. This global memory is cached in recent CUDA Compute Capability devices such as Fermi and Kepler but even if the register spills can be held within cache that is still significantly slower than real register memory, by about 15%.

In this case there was no difference in the registers required between the original and refactored ICP code and furthermore registers were not close to being a limiting factor for the occupancy of the SMXs.

Other resource pressure within the SMX can also lead to reduced occupancy so the next step was to check the occupancy for both codes with the Nsight profiler. It was observed that both codes used the same amount of shared memory and had the same block dimensions (i.e number of threads per block). Neither of the implementations had any resource factors which limited their occupancy of the SMXs and thus both codes were achieving 97% of the theoretical occupancy limit for the SMXs (which is a hardware imposed limit of the architecture, 2048 resident threads per SMX for Kepler).

This was quite surprising given the 40% disparity in performance between the two codes. Since both codes have such high occupancy rates it also seemed unlikely that either code was memory bound (as high occupancy covers memory access latency). An investigation of the memory accesses in terms of loads and stores and cache hit rates revealed both the codes had near identical memory profiles also.

The next area to investigate with the profiler was the branching behaviour of the code (e.g. for and conditional if statements). Branching itself is not a problem if all threads within a single warp take the same path. If the threads within a single warp diverge then a negative impact on performance is likely. As a warp is the smallest unit of scheduling within the GPU all arithmetic units executing a warp of 32 threads must execute the same instruction at the same clock cycle. When there is a divergent branch encountered within a warp the execution of the two paths after the branch is effectively serialised, i.e. all the threads pass all the instructions on both sides of the branch until the two distinct code paths split by the branch merge sometime later. The effect of the instructions is just masked off for whichever threads should not be executing the instructions based on the branch conditions.

In this case it was observed that the branching behaviour of both codes was not hampering performance and was identical. While both codes had over 80 branches taken and 20 not taken per warp the number of warps which had a divergence was less than 1%. This divergence is attributable to the cases where some individual values in the data depth maps from the Kinect may contain NaN values if the Kinect sensor was not able to retrieve a measurement for a a particular point in the depth map. The combinedKernel kernel checks for the NaN and returns immediately if found. Although the number of NaNs per depth frame is certainly larger than 1% a divergence does not occur if all 32 threads in a warp encounter an NaN which is likely to be the case more often than not as warps access contiguous elements of the depth map and areas which the sensor cannot retrieve values in tend to be clumped together rather than being individually scattered across the depth frame.

The final area to be investigated with the profiler was the instruction throughput. Both codes were performing almost identical absolute numbers of the various floating point instructions reported by the profiler which can be seen in Figure **??**

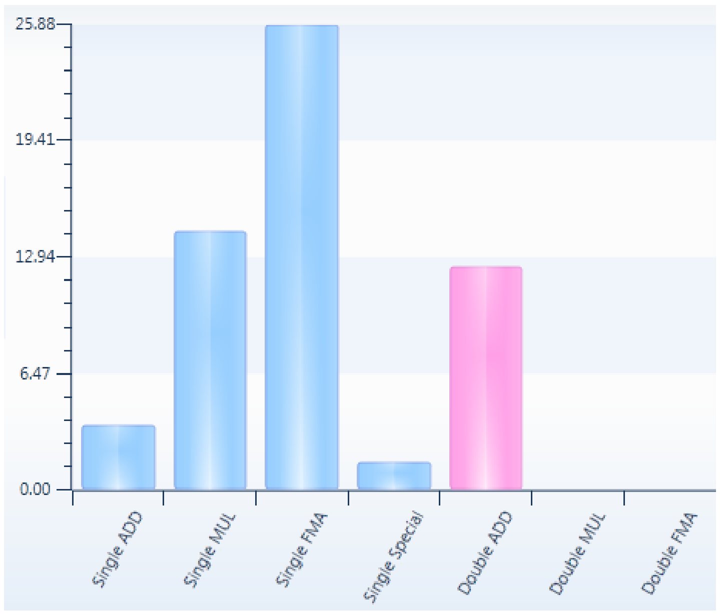


Figure 6.3: Absolute number (in millions) of floating point operations of various kinds reported by the Nsight compiler for a single invocation of the combinedKernel kernel.

The single special operations reported are the transcendental functions carried out by the special functions units within the SMX. In this case they are used in the calculation of the sin function as used in calculations involving the normal values within the data maps. All the raw data coming from the Kinect is 32 bit float data but double precision types are used to perform the reductions on the single precision data which explains their presence in the mix.

So although the number of floating point instructions being carried out by the two codes was near identical the throughput or FLOPs achieved by the two codes was markedly different as can be seen in table 6.1

	Single Precision	**Double Precision**
Original	29.03	7.94
Refactored	47.58	13.02

Table 6.1: The achieved GFLOPs for the original and refactored CUDA code in the ICP benchmarking application which represent a 63% performance increase for the a standalone invocation of the combinedKernel which is one part of the ICP loop.

At this stage of the profiling it was still unclear what was causing the disparity in performance between the two codes. They were performing almost identically in all metrics reported by the compiler except GFLOPs. The next step taken was to have the CUDA toolchain emit Nvidia's intermediate language PTX representation for of the source code kernels.

Upon comparing the two PTX codes they were identical except for a sequence of groups of instructions where the original code had a number of the PTX `cvt` instructions (used for type conversions) which were not present in the refactored code.

Listing 6.1: PTX from original code

```
//...
ld.volatile.shared.f64  %fd7, [%rl10+1024];
add.f64   %fd8, %fd30, %fd7;
cvt.rn.f32.f64  %f150, %fd8;
cvt.f64.f32   %fd30, %f150;
st.volatile.shared.f64  [%rl10], %fd30;
//...
```

Listing 6.2: PTX from refactored code

```
//...
ld.volatile.shared.f64  %fd7, [%rl10+1024];
add.f64   %fd22, %fd22, %fd7;
st.volatile.shared.f64  [%rl10], %fd22;
//...
```

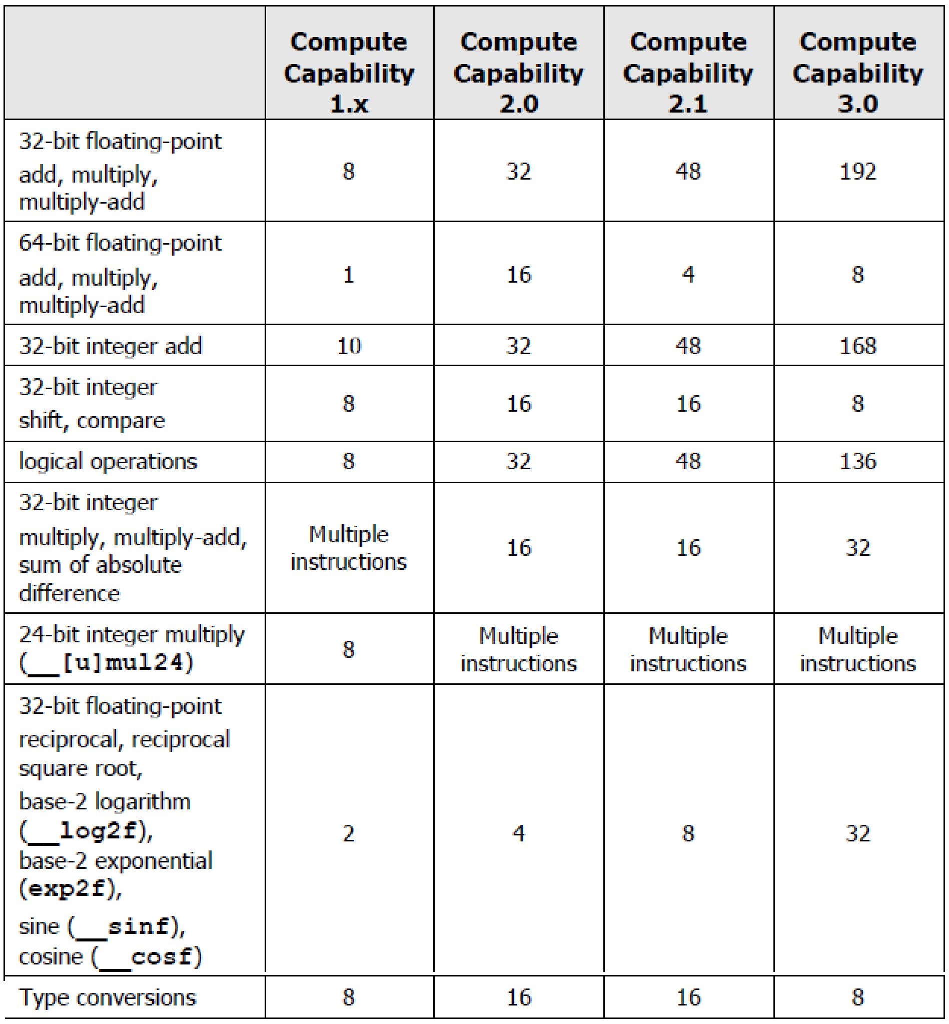
As can be seen the refactored code simply does a load, add and store in double precision. The original code however is performing the same load and add but is then doing a conversion on the result from double to single precision with rounding down and immediately afterwards converting the single precision value back to double precision before storing it.

This was quickly tracked down in the source code. In the original device code as present in the PCL library a functor named plus() is passed as an argument to a templatised implementation of a parallel reduction function operating on the GPU.

As mentioned earlier the raw data values coming from the Kinect are represented as single precision floats but the reduction is carried out in double precision. The plus functor written in the original library took two double values as arguments, added them and then returned a float which was immediately promoted to a double by an assignment in the reduction function.

When refactoring the original code to remove the functor (as the language feature is not provided in OpenCL) this problem was removed from the code, resulting in the the large increase in performance.

Once this improvement was introduced to the full scale KinFu application it provided a 30% boost in frame rate on an Nvidia Kepler GTX670 going from around 30FPS up to around 39FPS.



	Compute Capability 1.x	**Compute Capability 2.0**	**Compute Capability 2.1**	**Compute Capability 3.0**
32-bit floating-point add, multiply, multiply-add	8	32	48	192
64-bit floating-point add, multiply, multiply-add	1	16	4	8
32-bit integer add	10	32	48	168
32-bit integer shift, compare	8	16	16	8
logical operations	8	32	48	136
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	16	16	32
24-bit integer multiply (`__[u]mul24`)	8	Multiple instructions	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base-2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	4	8	32
Type conversions	8	16	16	8

Figure 6.4: Instruction throughput per clock for various types of instruction under different compute capabilities, referenced from NVidia CUDA C Programming Guide

The reason this instruction was so costly for performance can be understood by considering the information in Figure 6.4 regarding instruction throughput per clock (for a single SMX). As can be seen here the consumer grade Kepler chip used in the GTX680 can process 192 32bit floating point operations per clock in a single SMX. This corresponds 1:1 with the 192 CUDA cores present per SMX. For the type conversion instructions listed at the bottom of the figure we see that the same SMX can only process 8 type conversions per clock in a single SMX, 1/24th the rate it can do 32bit floating point math. AS is also clear from the table the gap between 32bit floating point performance and double precision math or type conversions has drastically widened from the previous generation of hardware (Fermi for example at Compute Capability 2.1). This is likely intentional from NVidia in an effort to persuade double precision users to move to the more expensive professional Tesla series cards which will have much better double precision performance.

While the Nsignt profiler makes clear the distinction between single and double precision arithmetic and special function instructions in its detailed breakdown it does not highlight the presence of type conversion instruction in device code. These types of instructions are clearly something to pay careful attention to as their presence can drastically reduce the performance of code on consumer grade GPUs.

6.3 CUDA versus OpenCL on Nvidia GPU for ICP

The performance results for the OpenCL port of the ICP code in the benchmarking harness showed a 30% performance drop moving from CUDA to OpenCL on Nvidia hardware. This is most likely down to NVidia's OpenCL compilation not producing binaries which are as well optimised as would be the case with CUDA code.

6.4 CUDA versus OpenCL on AMD GPU for ICP

When the ported OpenCL ICP code in the benchmarking harness was run on an AMD HD Radeon 7970, currently AMDs highest powered GPU the code performed much worse than on the Nvidia hardware. The results can be seen in Table 6.2

	OpenCL on Nvidia GTX 670	**OpenCL on AMD Radeon 7970**
Speedup	-30%	-140%

Table 6.2: The relative performance of the OpenCL port compared to the the CUDA code on the Nvidia GTX 670

This very poor performance for the flagship AMD Radeon card was unexpected. Upon profiling the code on the AMD GPU it was immediately clear what the issue was. AMD APP SDK's (latest version as of writing 2.7) OpenCL compiler was using 248 registers per thread for the combinedKernel kernel. This number is extremely high and in fact near the 256 register hardware limit for number of registers per kernel thread. In comparison the Nvidia's CUDA compiler was compiling the original kernel which was the source for the OpenCL port with the use of only 23 registers. This extremely high register useage by the AMD OpenCL compiler was causing a very low occupancy value for the Compute Units (CUs) on the AMD card. The AMD profiling tool reported occupancy of only 10% on the CUs, a figure which is much too low in order to cover the latency of memory transactions by the kernel.

Investigation was carried out to try to find the cause of such high register useage by the kernel but no solution was found in time for inclusion in this dissertation. As the GCN architecture is a revolutionary rather than evolutionary change in AMD's GPU design as discussed earlier and is also still very new to the market it is possible that this behaviour from the AMD OpenCL compiler is an anomaly which will be cleared up shortly. Scanning the AMD OpenCL developer forums many threads can be found discussing similar register related problems with the new GCN GPUs such as high register useage but also erratic register useage which changes by large amounts with very minor changes to source code. Several suggestion were tried to mitigate the problem, for example reducing the scope of private variables declared with the kernels by using braces to help the compiler reuse registers once variables went out of scope but this and some other experiments to reduce the high register useage had no effect.

As a last ditch resort to try to extract better performance from the HD7970 card a rewrite of the combinedKernel and its auxiliary device function search was planned and begun. The plan was to split the kernel in two and stage the results of the search kernel in global GPU memory and launching combinedKernel and the search separately on the host. This is obviously not ideal as it introduces what should be unnecessary kernel launches and global read/writes but if successful would likely offer an improvement. However further problems were encountered with AMD's OpenCL debugging tool which worked only intermittently when debugging device side code. Unfortunately NVidia's toolchain while allowing for the compilation of OpenCL code does not allow for debugging of it. A functioning implementation of this idea to decompose the kernel into multiple launches was not achieved in time for inclusion in the dissertation.

Chapter 7

Discussion, Further Work & Conclusions

This dissertation provided an opportunity for the author to gain experience in a variety of new areas. In terms of the domain of Computer Vision and 3D geometry and modelling this was a first introduction for the author to the area. With the explosion in popularity of the Kinect for Computer Vision applications we are already seeing other manufacturers release devices based on the same type of depth sensing technology. In the near future the choice and quality of such types of devices is only like to increase. The applications for everyday use of such technology for 3D modelling and augmented reality are only limited by ones imagination. Just this month Microsoft has submitted a patent application related to a Kinect Fusion like system in a mobile device [23]. With the rate of improvement in mobile SoCs incorporating data parallel GPUs it is only a matter of time until a system incorporating a depth sensor and enough processing power to perform realtime SLAM on the resulting data can be packaged into a mobile device. The advent of such a device could see people creating and uploading to the cloud 3D models of their surroundings just as easily as they do a photo or a video today. Certainly Computer Vision is one of the areas which can hope to benefit most from the revolutionary changes we are witnessing with the opening up of GPU processors to general purpose computation and this revolution which has been present on the desktop for a number of years is only just now beginning in the mobile device market. The opportunities for applying High Performance Computing techniques in Computer Vision only look set to increase and are likely to provide many exciting opportunities for those involved in each discipline.

One of the other main reasons this dissertation topic was chosen was for the opportunity to develop a deeper understanding of the technical background and skills required for GPGPU programming. While the taught portion of the MSc. provided a short introduction to this topic the dissertation provided the opportunity to learn more deeply about the variety of approaches for programming GPUs both in the proprietary world of CUDA (within which itself there are many for programmers now) and in the more open platform of OpenCL. Certainly at the moment OpenCL has a long way to go to compete with CUDA in terms of ease of use from a developer point of view. This disparity in ease of use between the two competing technologies runs right through the spectrum of activities from writing code to the build toolchain and profiling and debugging tools but also includes the wider community and ecosystem surrounding the developer. The value of an active user community of developers around a technology cannot be underestimated and NVidia seems to put a lot of effort into fostering this aspect of the CUDA platform. The value of the user community was brought into sharp focus when Nvidia suffered an attack on its registered developer website in the first week of July. While NVidia did make its developer tools available for download again within a fortnight the NVidia user forums have not yet been reopened and it was a pity not to have access to such a useful resource during the second half of the dissertation.

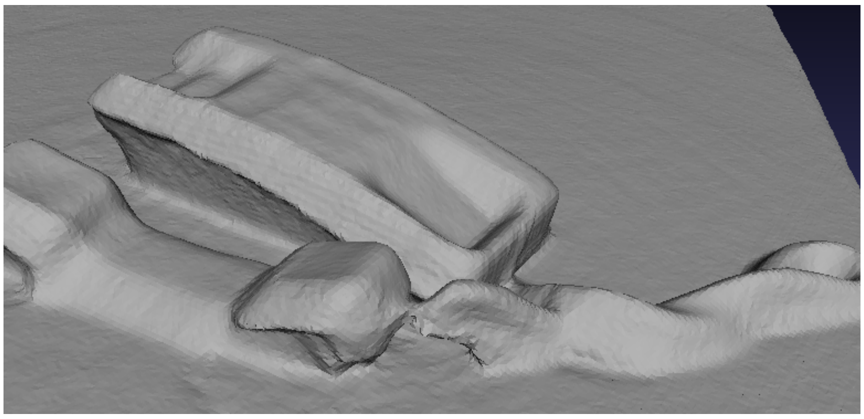


Figure 7.1: An image of a 3D mesh model extracted from a run of KinFu at Holoxica Ltd. on a telephone. This size of object is getting towards the extremely small end of the scale for KinFu's utility due to the Kinect Sensor's ability to resolve fine detail in depth maps. Future versions of the Kinect and competing products will likely show big improvements in this regard.

Despite the myriad advantages CUDA has it cannot be denied that OpenCL has a very significant advantage in its favour by being a cross platform and open standard. Naturally as with any open technology this means that development and evolution of the standard happens more slowly than with closed technologies but in the long term OpenCL may well close the gap on usability with CUDA. The difference in accessibility between the two is most keenly felt early in the early stages of familiarising oneself with GPU programming. Ironically in this respect CUDA can be a great asset for OpenCL in easing developers entry point to GPGPU computing. After becoming familiar with GPU programming with CUDA the move to OpenCL is then much more manageable than trying to master OpenCL from scratch.

7.1 Further Work & Conclusion

The opportunities for further work opened up by this dissertation are plentiful. Firstly I will contribute the change found resulting in a 30% performance increase back to the Open Source PCL library. Once the problems encountered with the OpenCL implementation of ICP on the AMD GPU are resolved continuing on the work to port the entire application to OpenCL is a worthwhile goal. Some work has been done by PCL already on texture and colour mapping the resulting 3D model produced by KinFu and it would be interesting to investigate the performance impact of this feature. Ultimately once a working OpenCL version of the system is achieved it would be very interesting to attempt to investigate the opportunities for splitting the application between the CPU and GPU rather than the current system which almost exclusively runs on the GPU.

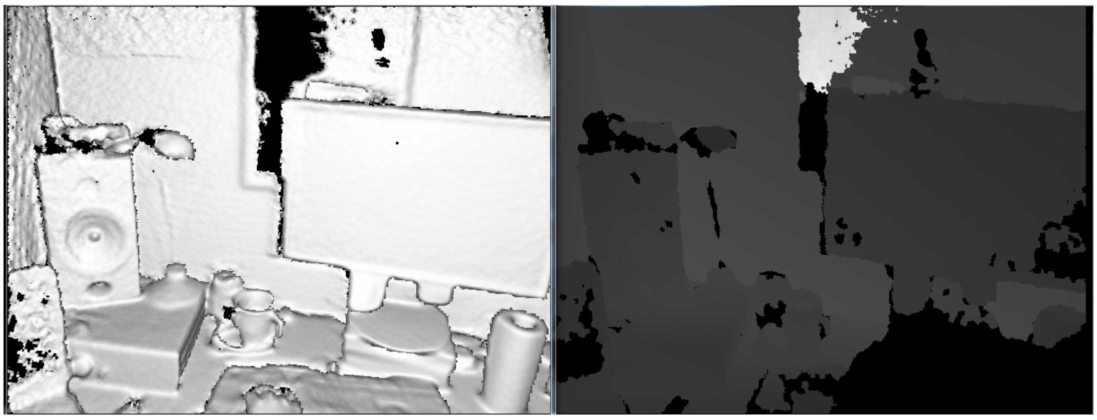


Figure 7.2: Screenshot of KinFu in action, (Raw depth map right side, 3D model left side). Here the application is delivering 38 to 40fps on an NVidia Kepler GTX 670 after the 30% performance gain due to the removal of the superfluous cvt instructions discovered in the CUDA PTX assembler code

In conclusion this dissertation has unearthed a significant performance increase of 30% in the existing open source PCL KinFu application which will deliver better performance for the many users and developers who are part of the PCL community and deliver a more robust closer to realtime performance for lower cost and laptop GPUs. On the desktop Nvidia GTX670 we are now seeing performance around 40FPS, well in advance of the 30FPS required to take full advantage of the Kinect. Furthermore we have begun the process of a fullscale port of the application to OpenCL by developing a functioning OpenCL version of the largest and most computationally expensive part of the application. Ultimately a full OpenCL port will allow for the use of a wider variety of hardware to run the system. The dissertation has also demonstrated that the as yet unreleased KinFu application from PCL is a system suitable for scanning small to medium sized objects or spaces in realtime with the Kinect. Ultimately Holoxica Ltd. can consider the KinFu application as a viable option for such scanning of real world objects for the construction of 3D holograms. The application will only improve further once it is officially released by PCL and gains a wider audience of users and developers. While there are also other interesting new software applications to consider for creating 3D models, for example 123DCatch, there is no other software available at the current time which can perform interactive 3D model construction in realtime in the manner of the KinFu system. Ultimately the realtime interactivity provided for 3D scanning by the KinFu application is its standout feature and what makes it such a compelling proposition.

Bibliography

[1] R. Szeliski. *Computer Vision: Algorithms and Applications.* Springer, first edition, 2010.

[2] P. Besl and N. McKay. *A method for registration of 3D shapes.* IEEE Trans. on Pattern Analysis & Machine Intelligence, 14:239-256, 1992.

[3] Y. Chen and G. Medioni. *Object modeling by registration of multiple range images.* Image and Vision Computing (IVC), 10(3):145-155, 1992.

[4] S. Rusinkiewicz and M. Levoy. *Efficient variants of the ICP algorithm.* 3D Digital Imaging and Modeling, Int. Conf. on, 0:145, 2001.

[5] B. Curless and M. Levoy. *A volumetric method for building complex models from range images.* ACM Trans. on Graphics, 1996.

[6] R. Newcombe et al. KinectFusion: *Real-time dense surface mapping and tracking.* In Proc. 10th IEEE Int. Symp. on Mixed and Augmented Reality, 2011.

[7] K. Lim Low, Linear Least-Squares Optimization for Point-to-Plane ICP Surface Registration, Technical Report TR04-004, Department of Computer Science, University of North Carolina at Chapel Hill, February 2004.

[8] Chris Angelini, Tom's Hardware. *GeForce GTX 680 2 GB Review: Kepler Sends Tahiti On Vacation.* http://www.tomshardware.com/reviews/geforce-gtx-680-review-benchmark,3161.html Retrieved 22/08/2012.

[9] Nvidia, CUDA C Best Practices Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf Retrieved 23/08/2012

[10] Vasily Volkov, Better Performance at Lower Occupancy, http://www.cs.berkeley.edu/ volkov/volkov10-GTC.pdf Retrieved 22/08/2012

[11] Nvidia, Geforce GTX680 Whitepaper. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf Retrieved 23/08/2012

[12] Nvidia, CUDA C Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf Retrieved 23/08/2012

[13] Nvidia, OpenCL Best Practices Guide. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf Retrieved 22/08/2012

[14] Nvidia, OpenCL Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf Retrieved 22/08/2012

[15] Nvidia, Using Inline PTX Assembly In CUDA. http://developer.download.nvidia.com/ compute/DevZone/docs/html/C/doc/Using_Inline_PTX_Assembly_In_CUDA.pdf Retrieved 24/08/2012

[16] AMD, OpenCL Programming Guide. http://developer.amd.com/ sdks/amdappsdk/assets/amd_accelerated_parallel_processing_opencl_programming_guide.pdf

[17] AMD, Graphics Core Net whitepaper.

[18] Programming Massively Parallel Processors: A Hands-on Approach 1st Morgan Kaufmann Publishers Inc. San Francisco, CA, USA Âl'2010 ISBN:0123814723 9780123814722

[19] CUDA Application Design and Development, Rob Farber. Morgan Kaufman 2011

[20] Ryan Smith, AnandTech. *The Kepler Architecture: Fermi Distilled.* http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/2 Retrieved 22/08/2012.

[21] Ryan Smit, AnandTech. AMD Radeon HD 7970 Review: 28nm And Graphics Core Next, Together As One http://www.anandtech.com/show/5261/amd-radeon-hd-7970-review Retrieved 23/08/2012

[22] Radu Bogdan Rusu and Steve Cousins,3D is here: Point Cloud Library (PCL), IEEE International Conference on Robotics and Automation (ICRA), 2011, Shanghai, China

[23] Microsoft patent applications take Kinect into mobile cameras, movie-making http://www.engadget.com/2012/08/02/microsoft-patent-applications-take-kinect-into-mobile-cameras/