Department of Computing, Imperial College London

Final Year Project

Investigating the Data Flow and Parallelism in Real-Time Dense SLAM

Daniel Woffinden <daw10@ic.ac.uk>

Supervisor
Prof. Paul Kelly

Second Marker
Prof. Wayne Luk

17th June 2014

Abstract

As computing devices become more powerful, increasingly striving to understand the physical world, computer vision will be a key driver for portable multicore computing.

We analyse a representative dense SLAM implementation, KFusion, and investigate the dependencies and data flows between its constituent tasks. We create a new performance benchmark based upon KFusion, in collaboration with members of the PAMELA project.

We use Qualcomm MARE to create an efficient multicore version of KFusion, exploiting data parallelism on the CPU. We evaluate MARE for robot vision applications, and provide insight to guide future research on exploiting task parallelism in KFusion and other such applications.

Investigating the scalability of our implementation, we find that the tracking phase becomes a bottleneck as we attempt to exploit many parallel cores with KFusion. We suggest possible solutions to increase multicore tracking performance, as well as points of task parallelism that could be leveraged when constrained by real-time deadlines.

Acknowledgements

I would like to thank:

- My supervisor, Prof. Paul Kelly, for providing the inspiration for this project, and invaluable advice and assistance throughout.
- Dr. Luigi Nardi, for the help and support he provided throughout, as I battled with KFusion.
- Gerhard Reitmayr, for creating a free and open-source implementation of Kinect Fusion.
- PAMELA project members, particularly John Mawer and Andrew Nisbet for their work on the C++ version of KFusion.
- Qualcomm, for creating and supporting MARE, in particular Dr. Călin Caşcaval, for his support and feedback.
- Renato Salas-Moreno, for his insight into KinectFusion, and help with OpenNI2.
- Jan Jachnik and Niklas Hambüchen, for their code contributions that helped to shape the OpenNI2 driver.

Contents

List of Figures

List of Tables

Listings

1. Introduction

One of the biggest drivers for future multicore computing will be computer vision. In particular, equipping robots, sensors and wearable devices with 3D scene understanding will become a platform for a huge range of applications. A key issue is optimisation for power, as well as performance.

Existing applications make extensive use of hardware parallelism for data parallel tasks – GPU acceleration is commonly employed to accelerate the processing of particular sub-tasks in the pipeline. This approach is easy to reason about, and permits a high degree of code reuse.

We investigate the dependencies between these tasks for KFusion, a representative dense SLAM implementation, with the aim of increasing performance and efficiency by dynamically exploiting multicore and manycore hardware.

1.1. Structure

In chapter 2, we introduce Qualcomm MARE (section 2.1) and give a brief overview of data and task parallel programming (section 2.2). Section 2.3 then introduces the context for KFusion, giving a broad intuition of the SLAM problem, and the KinectFusion algorithm.

In chapter 3, we conduct a thorough analysis of the structure of the KFusion application, explaining its composition of tasks and data dependencies between them. Chapter 4 then explains how we exploit this in our implementation, and contains our performance analysis, detailing the scalability of our solution, comparing it to the existing serial C++ implementation, and and OpenMP implementation.

1.2. Motivation

Vision algorithms are becoming increasingly sophisticated, capable and performance intensive. While commodity hardware continues to grow in power and affordability, this is increasingly due to parallelism, rather than single-threaded performance.

The desire to employ computer vision on low cost hardware or mobile platforms requires us to exploit this parallelism fully, to maximise performance and/or reduce power consumption.

With a highly diverse hardware landscape and careful optimisation necessary to squeeze performance out of a platform, attaining *performance portability* of an implementation, such that it can perform efficiently on multiple platforms, can be hard to achieve.

Many algorithms, such as Kinect Fusion (see section 2.3.2), are static and sequential, exploiting data parallelism to accelerate key computations. While capable of high performance and relative simplicity, this approach is inflexible, leaving the algorithm unable to adapt to fully exploit the hardware upon which it is running.

1.3. Approach

We adapt the real-time KFusion to be suitable as a reliable, off-line performance benchmark.

We perform a qualitative analysis of the structure of KFusion, the data dependencies between its tasks and the scope for task and data parallelism, and provide insight to guide further research.

We use Qualcomm MARE to exploit data parallelism within KFusion, and perform a quantitative analysis on its performance.

1.4. Contributions

The major contributions in this work were:

- Creating a new off-line dense SLAM performance benchmark based upon KFusion, in collaboration with Luigi Nardi and other PAMELA project members.
- Creating an efficient multicore implementation of KFusion using Qualcomm MARE. We evaluated both the usability and performance of MARE, a relatively new library, in this use case.
- Investigating the scalability of our implementation, providing insight into the performance bottlenecks in KFusion that limit our utilisation of multicore hardware.
- A quantitative and qualitative analysis of the performance and data flow structure of KFusion, and of its potential for both task and data parallelism. We provide insight into likely targets for future work exploiting the task parallelism in KFusion, and in similar computer vision applications, for performance and energy efficiency.

2. Background

In this chapter we compare task-based parallelism to the *data* parallelism supported by many prevalent, general purpose parallelism frameworks. We look at some of the important distinguishing features of Qualcomm Multicore Asynchronous Runtime Environment (MARE) and similar projects, and discuss the advantages and trade-offs of these approaches.

2.1. Qualcomm MARE

MARE (Multicore Asynchronous Runtime Environment) [1] is a C++ library, designed by Qualcomm Inc. to allow application developers to exploit parallelism on multicore smartphones and tablets. However, as a powerful, flexible, portable and performant parallel programming library, MARE has a wide range of potential applications, and is also well suited to general purpose, and high-performance computing.

MARE is built upon the concept of task graphs (visited in section 2.2.3), and provides high-level APIs which facilitate other styles of parallel programming, by mapping them onto MARE's task model.

2.1.1. Synchronous Data Flow (SDF)

The latest version of MARE contains the MARE SDF API for Synchronous Data Flow programming [8, p. 26]. The SDF API allows users to describe a graph of task nodes, with a C++ node-function on each, and channels between them. Each channel acts as a FIFO queue, where producer nodes push data items, and consumer nodes are executed as soon as there is data in each input channel, popping one item from each.

This model permits automatic parallelisation and pipelining of independent tasks, while maintaining synchronous data-flow between dependent tasks, but requires sweeping changes to an application's structure to be put to best use.

2.1.2. Heterogeneous Computing

MARE 0.11.0 adds to its abstractions the ability to schedule OpenCL kernels for execution on the GPU, taking away the need to worry about execution models details from the programmer and allowing an easier mixture of CPU and GPU code [9, p. 6][8, p. 54].

MARE provides buffers than can automatically copy data between the CPU and GPU when needed, making mixed, heterogeneous programming much simpler.

[1] `https://developer.qualcomm.com/mobile-development/maximize-hardware/parallel-computing-mare`

2.2. Forms of Parallelisation

2.2.1. Data Parallelism

Data parallel algorithms are those whose parallelism comes from simultaneous operations performed across large sets of data [6]. Elements are distributed between processor nodes, and each node performs the same operation on different data elements in parallel.

Data parallel programming models are used to accelerate a wide spectrum of algorithms, and are supported by a plethora of hardware architectures, programming languages and software tools. The vast majority of x86 microprocessors sold today support a large set of Streaming SIMD Extensions (SSE) instructions, added to improve the performance of multimedia applications with data parallel arithmetic [14, p. 649].

Data parallel problem decomposition forms the basis of effective Graphics Processing Unit (GPU) acceleration - large arrays of result data can be partitioned into blocks, and then again into elements, forming fine-grained computations that can be executed in parallel [14, p. A-17].

Modern software tools can make parallelising data parallel tasks conceptually very simple - OpenMP allows the parallel execution of different iterations of a loop with a simple pragma, such as in Listing 2.1 [13]. Collection libraries such as ScalaBlitz permit very convenient parallel operations on collections of data, e.g. Listing 2.2 [15].

```
1 const int N = 100000;
2 int a[N];
3
4 #pragma omp parallel for
5 for (int i = 0; i < N; i++) {
6   a[i] = 2 * i;
7 }
```

Listing 2.1: An OpenMP parallel for loop

```
1 import scala.collection.par._
2 import Scheduler.Implicits.global
3
4 val sum = (0 to 15000000).toPar.reduce(_ + _)
```

Listing 2.2: A ScalaBlitz parallel reduction, from `http://scala-blitz.github.io/`.

2.2.2. Task Parallelism

Where data parallel systems focus on the division of data in order to partition the computation for parallel execution, task parallelism focuses directly on division of execution processes.

The most basic form of task parallelism is that of manually managed threads - the programmer explicitly creates threads managed by the Operating System (OS). Each thread essentially runs its own program, and careful reasoning is required to write multithreaded applications in this way.

As with data parallelism, modern technologies can provide us higher levels of abstraction over explicit thread management.

Java's `ThreadPoolExecutor`[2], in the `java.util.concurrent` package, for example, greatly reduces the need for explicit thread management in Java. Instead of creating a single OS thread per logical worker, a `ThreadPoolExecutor` serves as a *thread pool*, with one thread per logical processor. Tasks are then written as objects implementing the `Runnable` interface, and can then be passed to the `ExecutorService`. The service executes all queued tasks in turn, one per thread. When the task being executed on a thread completes, the thread will retrieve another from a shared queue, running that to completion as well. Listing 2.3 shows an example usage of this, scheduling ten tasks to be executed using as many threads as the underlying processor has cores.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import static java.util.concurrent.Executors.newFixedThreadPool;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService pool =
            newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        for (int i = 0; i < 10; i++) {
            final int num = i;
            pool.submit(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(100);
                        System.out.println("Executing Runnable #" + num + "...");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
        pool.shutdown();
        pool.awaitTermination(200, TimeUnit.MILLISECONDS);
    }
}
```

Listing 2.3: An example of parallel tasks being executed by a Java thread pool.

Apple's Grand Central Dispatch (GCD), a task parallelism technology for Mac OS X and iOS [4], is similar to `ThreadPoolExecutor`, being a thread pool based task scheduler, but offers several features not found in the `java.util.concurrent` offerings. As well as scheduling them individually, GCD allows tasks to be arranged into 'Dispatch Groups', and clients can wait for a particular group of tasks to complete.

[2] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html`

GCD also allows the automatic scheduling of a task for every element of an array - Listing 2.4 shows a simple for loop; by turning the body into a block (Listing 2.5), it is scheduled to be run `count` times, once for each array index, *in parallel* (example from [3]). This gives us a very convenient facility to parallelise loops, as we could with OpenMP, while also allowing more general task based parallelism, all sharing the same global work queue for efficiency.

```
for (i = 0; i < count; i++) {
      results[i] = do_work(data, i);
}
total = summarize(results, count);
```

Listing 2.4: A serial for loop.

```
dispatch_apply(count, dispatch_get_global_queue(0, 0), ^(size_t i){
     results[i] = do_work(data, i);
    });
total = summarize(results, count);
```

Listing 2.5: The same loop, with each iteration executed in parallel with GCD.

2.2.3. Task Graphs

Task parallelism, as we saw in section 2.2.2, allows us to divide our computation into sub-tasks, and schedule them for concurrent execution. The problem with this model, however, is that for the tasks to be executed simultaneously, they must be independent. Such a requirement is reasonable when, for example, we wish to parallelise a single for loop without waiting for the result. If, however, we wanted to represent, an entire program as a set of tasks, there will inevitably be *dependencies* between the tasks, imposing a partial ordering on the computations.

Unlike GCD and `java.util.concurrent`, which operate on *queues* of tasks, Qualcomm's MARE operates on *graphs* of tasks. This approach allows us to additionally encode task dependencies, and ensure that a task is scheduled only after the tasks that it depends on have completed.

Similar to GCD, MARE provides helpers, known as 'patterns', to easily express data parallel operations such as loop traversal – with a range iterator and an anonymous function, as in Listing 2.7 [8, p. 25] – breaking it into tasks to be executed simultaneously. Note that MARE does not create a task for every iteration of the loop, but uses a heuristic method to group several iterations into a task, to reduce the overheads associated with task management, and improve locality [8].

This is similar to OpenMP's `#pragma omp task` directive, but offers much easyier to use, general purpose dependencies, rather than OpenMP's memory access synchronisation between dependent sibling tasks.

[3] `https://en.wikipedia.org/wiki/Grand_Central_Dispatch?oldid=595238699#Examples`

```
#include <mare/mare.h>
#include <stdio.h>

int main() {
    mare::runtime::init();

    auto hello = mare::create_task([]{
        printf("Hello ");
    });
    auto world = mare::create_task([]{
        printf("World!\n");
    });

    // Make sure that "World!" prints after "Hello"
    hello >> world;

    mare::launch(hello);
    mare::launch(world);

    mare::wait_for(world);
}
```

Listing 2.6: A simple MARE program that prints 'Hello World!' using two tasks and a single dependency. Based on examples from [11] and [8].

Dependencies

If a task `t2` is declared to be dependent on `t1`, either with `mare::after(t1, t1)` or `t1 >> t2`, then 'the MARE runtime guarantees that `t2` does not begin execution until `t1` completes execution, regardless of how many hardware execution contexts are available in the system.' ([8]) Because the source for MARE is not available, it is not possible to investigate exactly how this is implemented.

As tasks in MARE do not send values to each other (as opposed to, e.g., Cilk threads [1]), there are no *data* dependencies, and therefore no additional points of implicit synchronisation between tasks.

Explicit synchronisation is of course possible however, and MARE provides `mare::barriers`, upon which multiple tasks can `wait()` and synchronise. `mare::condition_variables` facilitate inter-task communication through `wait()` and `notify()`, etc., without blocking the MARE scheduler [8].

Access to Shared Data

As MARE tasks are arbitrary functions, executed on threads in a shared process address space, they have access to all shared process data, making data races a possibility.

Such practices are of course best avoided when programming parallel tasks. While reasonably effective data race detection tools are available, MARE's dynamic nature makes automatic analysis more difficult.

```
#include <mare/mare.h>
#include <mare/patterns.hh>

int main() {
  mare::runtime::init();

  std::vector<int> vin(1024);
  std::vector<int> vout(vin.size());

  mare::pfor_each(size_t(0), vin.size(), [=,&vin,&vout] (size_t i) {
    vout[i] = 2*vin[i];
  });
}
```

Listing 2.7: A parallel for loop expressed with MARE. Based on an example in [8, p. 25].

MARE attempts to sidestep these issues by encouraging data dependencies to be explicitly declared, so that tasks can be scheduled safely. MARE also supporting *task-local* data storage or, when appropriate, *thread-local* or *scheduler-local* storage.

Task Scheduling

When a task is executed, it is assigned to a thread, and it will stay assigned to the same thread until it either completes, or reaches a 'safe point' [8]. Safe points are defined as MARE API calls where the thread upon which a task executes before the call may not be the same as the thread upon which it executes after the call [8]. In the current MARE release, the only safe points are the `mare::wait_for()` methods [8]. Of course it is still possible for a task to temporarily yield to the scheduler, and another task allowed to execute, but as long as it did not reach a safe point or complete, it will resume on the same thread.

This is opposed to the work-stealing policy employed in Cilk, where a processor which runs out of work is able to steal ready threads from a working processor's spawn tree [1]. This approach leads to a more balanced work distribution, however requires much more careful synchronisation of these data structures, and incurs a greater overhead during context switches and work queuing.

2.3. KFusion

In this section, we introduce the KFusion application that will be the focus of this work, as well as providing a brief overview of the SLAM problem, and of the KinectFusion algorithm, of which KFusion is an implementation.

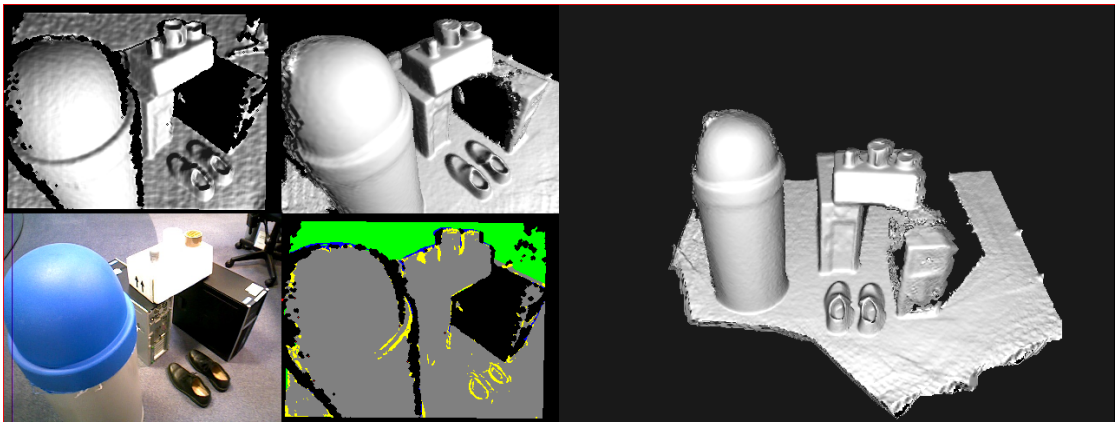


Figure 2.1.: A screenshot of KFusion processing the `weird.oni` recording. The images represent, clockwise from the top-left, the raw depth input, the estimation of the scene from the position of the camera (Raycast), the estimation of the scene from a user-configurable position (Render), the ICP outliers from tracking and the raw RGB image, which is otherwise unused.

2.3.1. Simultaneous Localisation and Mapping (SLAM)

SLAM is a fundamental problem in robotics, where a robot must acquire a map of its environment while simultaneously localising itself relative to that map [18, p. 309]. Many approaches to SLAM exist. Sparse feature filtering systems like MonoSLAM [3] have been steadily improved over the years, but a recent breakthrough has been the use of 'dense SLAM' methods to reconstruct and track full surface models, represented as meshes or implicit surfaces, using commodity GPU hardware and depth cameras [17].

2.3.2. KinectFusion

KinectFusion is a system for accurate real-time mapping of complex indoor scenes, using a mobile depth camera and commodity GPUs. A 'dense SLAM' system, it fuses depth data streamed from a Kinect sensor into a single three-dimensional voxel volume, and constructs a global implicit surface model of the observed scene in real-time [12].

From [10], the heart of KinectFusion is a pipeline consisting of four main stages:

1. Depth Map Conversion. The live depth map is converted from image coordinates into 3D points (referred to as vertices) and normals in the coordinate space of the camera.
2. Camera Tracking. In the tracking phase, a rigid 6DOF transform is computed to closely align the current oriented points with the previous frame, using a GPU implementation of the Iterative Closest Point (ICP) algorithm…Relative transforms are incrementally applied to a single transform that defines the global pose of the Kinect.

3. Volumetric Integration. Instead of fusing point clouds or creating a mesh, we use a volumetric surface representation…Given the global pose of the camera, oriented points are converted into global coordinates, and a single 3D voxel grid is updated. Each voxel stores a running average of its distance to the assumed position of a physical surface.
4. Raycasting. Finally, the volume is raycast to extract views of the implicit surface, for rendering to the user. When using the global pose of the camera, this raycasted view of the volume also equates to a synthetic depth map, which can be used as a less noisy more globally consistent reference frame for the next iteration of ICP. This allows tracking by aligning the current live depth map with our less noisy raycasted view of the model, as opposed to using only the live depth maps frame-to-frame.

(Izadi et al. [10])

KinectFusion is a relatively simple, yet powerful dense SLAM algorithm. As is typical with many vision algorithms, is composed of many smaller tasks, making it an excellent subject for studying potential general optimisation techniques.

2.3.3. KFusion

The original KinectFusion was developed at Microsoft Research [12], and the source code of the original implementation described is not available.

A free and open source reimplementation of KinectFusion as described in [12], KFusion, was written by Gerhard Reitmayr with contributions from Hartmut Seichter, and made available under the MIT license [16]. KFusion is written in C++, with the GPU accelerated kernels in CUDA [4], and is capable of interfacing with the Microsoft Kinect sensor using either the Microsoft Kinect Software Development Kit (SDK) [5] or the open source libfreenect library [6].

Other implementations exist, such as the Point Cloud Library's KinFu [7], but as that implementation is part of a much larger framework and under active development, we felt that KFusion was better suited to isolated study.

2.3.4. PAMELA KFusion

For this work, we will focus on a fork of the Reitmayr KFusion code, developed by the PAMELA group [8]. The PAMELA fork adds a number of features useful for this work:

- Support for a wider selection of camera devices, through an interface to the OpenNI2 library [9].

[4] `https://developer.nvidia.com/cuda-zone`
[5] `https://www.microsoft.com/en-us/kinectforwindowsdev/`
[6] `http://openkinect.org`
[7] `http://pointclouds.org/news/2011/12/08/kinectfusion-open-source/`
[8] `http://apt.cs.manchester.ac.uk/projects/PAMELA/`
[9] At the time of writing (2014-05-14), `OpenNI.org` is no longer available, but the source can be found at `https://github.com/OpenNI/OpenNI2`, and `http://structure.io/openni` maintains a mirror and fork of the code

- Support for recording camera output to a file, and playing back such a file without a camera (the `-of <file>` and `-if <file>` flags respectively).
- The ability to dump the internal integration volume to a file, allowing us to compare the internal representation used by different versions of KFusion (the `-ov <file>` flag).
- An OpenCL implementation of KFusion, a single-threaded C++ implementation that eschews all GPU acceleration, and later an OpenMP-accelerated implementation. These are chosen with the `cmake` flag `-DLANGUAGE=<cuda|opencl|cpp|openmp>`.

See section 3.4 for more details on using PAMELA KFusion as the basis for a repeatable benchmark.

2.4. Summary

We have provided a brief introduction to parallelisation, namely data, task and task-graph parallelism; and investigated some design decisions made by a number of existing projects, frameworks and standards.

We have shown the main benefits and drawbacks of task-based parallelism and explained why we have chosen to investigate its potential, and in particular our choice to study KFusion. We provided a brief overview of the KFusion algorithm, and some context for the SLAM problem

In chapter 3, we present an analysis of KFusion, and it's suitability for these techniques and technologies, before exploiting data parallelism and analysing the performance in chapter 4.

3. Analysis of KFusion

In this chapter we present an analysis of the KFusion implementation, detailing the structure of the algorithm, the coarse tasks into which it can be broken, and the data dependencies between these tasks. We investigate the potential for parallelising these tasks with MARE, and predict the modifications that would have to be made for various techniques to be applied successfully.

In section 3.4, we detail the necessary steps to transform KFusion from a real-time, live-input, user-facing application into one suited to repeatable performance experiments.

3.1. Algorithm Structure

Pipelining is a technique near-universal in hardware, that increases processor throughput by overlapping the execution of multiple instructions [14, p. 330]. It was initially hoped that the KFusion algorithm could be pipelined, extracting additional parallelism and flexibility by executing multiple stages at once, or in a time-sliced fashion. Further analysis, however, shows that much of the algorithm forms a tight cycle, as shown in Fig. 3.1.

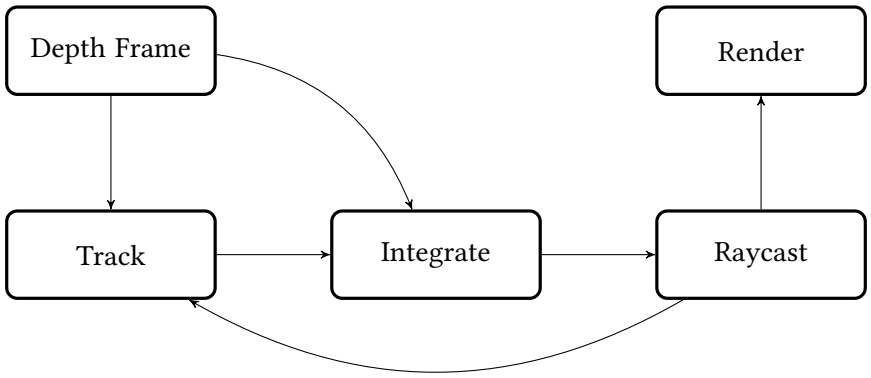


Figure 3.1.: The structure of the KFusion algorithm, showing simplified data flow.

The tracking stage of the KinectFusion algorithm relies upon an up-to-date raycast against the current integration volume, using the previous pose estimate to update accurately. Although introducing delay to increase parallelism was not investigated, it is expected that such an approach would increase the computational cost per frame of tracking, and reduce accuracy – similarly to using a higher capture rate and producing a lower quality image, as investigated by Handa et al. [5].

3.2. Data Flow

KFusion uses many buffers to facilitate the passing of data between stages of the algorithm, these relationships are outlined in Fig. 3.2.

Most buffers, such as `rawDepth`, are single image buffers which are overwritten on each iteration. By contrast, `pose` is a single `Matrix4`, which is passed around by value, and `integration` is the persistent 3D volume, which is updated (by default) every other iteration.

Not pictured, there are also many intermediate buffers used within the Track stage.

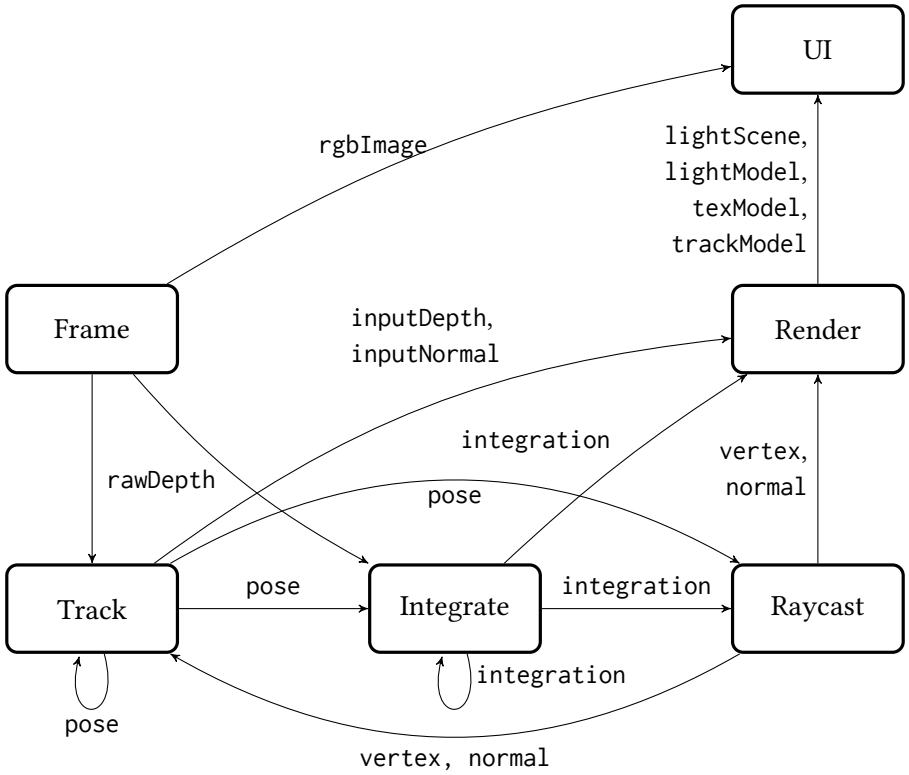


Figure 3.2.: Data flow and dependencies within KFusion. Arc labels refer to shared buffers and loops signify that the output of one iteration of a stage is used as input to the next iteration.

3.3. Scope for Parallelism

3.3.1. Data Parallelism

Initially designed for acceleration with CUDA, the KinectFusion algorithm contains much opportunity for data parallelism within each stage of the algorithm. Many of the kernels were invoked for parallel execution of thousands of independent computations, e.g. for every pixel

in an image buffer. This code was converted to C++ as two dimensional for-loops, and this parallelism is easily exploited in MARE (see chapter 4) or similar frameworks.

3.3.2. Task Parallelism

We had initially hoped to be able to pipeline the KFusion algorithm, executing each stage in parallel, but for different iterations. The tight Track-Raycast feedback loop makes this approach unwise, however. While an outdated scene estimation from the raycasting would likely suffice for tracking, the performance would be reduced – reducing accuracy, increasing the processing time for the track stage and increasing the risk that tracking is lost altogether (the greater the difference between the frames, the more iterations of ICP are required).

This dependency makes full pipelining undesirable, some asynchronous processing could be possible. It could be possible to 'peel off' some of the frame preprocessing and rendering stages, executing them asynchronously:

- While the core of the Track phase, `track` and `reduce`, require the output of Raycast, the preprocessing steps `bilateral_filter`, `halfSampleRobust`, `depth2vertex` and `vertex2normal` could in theory be executed as soon as the next frame is available, potentially while Raycast is still executing for the previous frame, reducing the delay before tracking can begin.

- The Render stage of KFusion consists of several independent tasks, some solely for the UI. While some of these tasks require the output of Raycast, KFusion also executes a second raycast which could be run in parallel as soon as Integrate has finished, and tracking does not require that this has completed. Other tasks simply involve moving data around from the frame acquisition and tracking stages, and could be started as soon as this data is available.

While there is clearly scope for exploiting task parallelism here, there are a great number of data buffers which would have to be guarded against concurrent access. This would likely involve double-buffering to permit concurrent updates of the buffers while tasks are still executing on the old contents, increasing memory consumption and synchronisation overhead, for unclear performance benefits.

It could be possible to use MARE's support for SDF graphs to manage this parallelism for us, but this would involve rewriting much of KFusion as separate task nodes, and we would require transparent reuse of multiple intermediate buffers, again increasing memory usage.

3.3.3. Integration

While up-to-date tracking information is essential to the operation of KFusion, it is much less important that all depth data be actually integrated into the volume. KFusion by default skips the integration of every other frame of data, and this rate is customisable through the `integration_rate` variable. As the single most computationally expensive part of the algorithm, omitting the integration step greatly reduces the processing time of a frame, and by omitting some frames, KFusion is better able to maintain a high tracking frame rate, while sacrificing negligible accuracy.

In theory, for a small target area, once KFusion has built a complete map, integration becomes unnecessary, and the algorithm could continue to track with high accuracy while using much less power by reducing the integration rate, or eliminating it entirely.

Since the KinectFusion algorithm is clearly tolerant of unintegrated data, another possibility would be to perform the integration asynchronously, alongside the tracking loop. This would mean that raycasting may be performed on a volume that has been partially updated, but if this is acceptable, then it allows tracking to be performed at a consistent, high frame rate, while integration happens at a flexible rate, exploiting spare cycles wherever possible, and the rate can even be varied as the situation demands.

3.3.4. Heterogeneous Compute

MARE has support for heterogeneous computing, through embedding OpenCL kernels into a program, and mixing them with standard C++ tasks. It manages data using a buffer implementation that can copy data between the CPU and the GPU when it is needed. This approach has the potential to allow us to fully utilise both the Central Processing Unit (CPU) and the GPU, for maximum performance, but would require all KFusion data to be held in these buffers, and the merging of the OpenCL and the MARE code.

In theory, by using each device for the kernels to which it is best suited, we can maximise performance. However, the overhead of copying large amounts of data, e.g. the integration volume, between the CPU and the GPU would be detrimental to performance, so care must be taken to split the work at a point in the algorithm that minimises such data transfer. Such a split could be to perform tracking on one device, and integration, raycasting and rendering on the other, as the tracking stage does not require the integration volume.

3.4. Creating a Benchmark from KFusion

KFusion is designed as a real-time application – frames are read directly from a camera, processed, and the state is displayed to the user, until they quit the program. One of the key objectives of the PAMELA fork of KFusion is to facilitate performance testing of KFusion, and the code contains several features to that end.

- After our addition of an OpenNI2 back-end to KFusion, support was added in the PAMELA version for recording and playback of camera streams, through the `-of <file>` and `-if <file>` options respectively. This allows test cases to be prerecorded, and KFusion can be ran on them reproducibly, without a camera.
- By default, OpenNI2 will play back a recording at the rate it was captured, by default 30 fps. It will also skip frames when necessary to keep pace, if the playback falls below 30 fps. The addition of a `-f` flag to KFusion allows us to force OpenNI2 to deliver every frame from the recording, and to do so with no deliberate delays.
- As a real-time application, the main operation of KFusion is tied to the visual output produced for the user, with `compute()` being called in the main GLUT loop when a screen refresh is required. This is fine for normal operation, however it can create unpredictable

delays between the processing of frames, interfering with measurements, as well as requiring an OpenGL window, and capping playback at the refresh rate of the user's monitor (typically 60 Hz).

The PAMELA fork of KFusion provides the `TERMUI` CMake variable, which when set (e.g. `cmake -DTERMUI=1`) completely disables all OpenGL and GLUT code, permitting measurement at maximum performance.

It is pointless performing performance comparisons between various implementations of an algorithm, if we do not also ensure functional correctness. The `-ov <file>` KFusion flag causes the integration volume at the heart of KFusion to be written to a file when it exits. This can then be compared against the volume produced by an alternate version with the `tools/ABTesting.cpp` tool, developed by Luigi Nardi, to ensure agreement.

The volumes were initially compared by comparing the infinity norm – the maximum difference between corresponding, non-nil voxels in each volume – against a given threshold. This turned out to be a bad metric, as small differences in, e.g. tracking, can leave a very small number of cells, such as those representing the edge of an object, with drastically different values, despite the volumes appearing almost identical.

The representation eventually used to compare volumes was an 'error histogram', which places the difference between each corresponding pair of cells into a bucket by its magnitude, and allows volumes to pass the equivalence test with large numbers of very small differences or very small numbers of large differences.

3.5. Summary

We have shown that, as it was initially developed as a CUDA application, KFusion contains much scope for data parallelism, which goes unexploited in the CPU implementation. Chapter 4 will detail our efforts to do so in a new MARE implementation.

While there is some scope for task parallelism in KFusion, the need for a close-feedback tracking loop greatly lowers our flexibility to break apart the pipeline. Stages before and after tracking could in theory be peeled off and executed asynchronously, but there exist a great number of data dependencies between tasks, particularly those serving the UI, which make this difficult.

4. Exploiting Data Parallelism in KFusion

In this chapter, we create a MARE implementation of KFusion in order to investigate the data parallelism present, exploiting it on the CPU in a similar way to how the CUDA version runs on the GPU.

We assess the scalability of our MARE implementation in order to better understand the performance bottlenecks in KFusion, and its performance profile will help to guide future optimisation efforts.

By assessing the limits to parallelisation, and the scalability of KFusion's various stages in the face of increasing system resources, we will better understand how those resources should be allocated for increased performance and efficiency.

4.1. Approach

Because it was adapted from the CUDA version of KFusion, in many places the C++ implementation has for-loops where the CUDA version contained parallel kernel invocations.

These for-loops present the same scope for parallelisation, often per-pixel, and we can exploit the independence of each calculation, they can be executed concurrently across processor cores.

Using MARE's `pfor_each(...)` construct, these loops were easily and efficiently parallelised. An example of this transformation is shown in Listing 4.1, a two-dimensional for-loop which, when parallelised, would resemble Listing 4.2.

```
1 for (int y = 0; y < normal.size.y; y++)
2   for (int x = 0; x < normal.size.x; x++)
3     renderLightKernel(make_uint2(x, y), out, vertex, normal, light, ambient);
```

Listing 4.1: A typical per-pixel for loop in the C++ version of KFusion.

```
1 mare::pfor_each(0u, normal.size.y, [&](unsigned int y) {
2   for (int x = 0; x < normal.size.x; x++)
3     renderLightKernel(make_uint2(x, y), out, vertex, normal, light, ambient);
4 });
```

Listing 4.2: The per-pixel loop adapted for parallel execution with MARE.

Not all kernels are so obviously parallel, however. In some cases, the transformation from CUDA code resulted in variables being shared across iterations, which were moved back into the body of the loop. Other cases were less obvious than a per-pixel loop, where iteration was

performed in explicit blocks in CUDA for efficiency, but the transformation to parallel MARE was conceptually similar.

4.2. Performance Evaluation

Multicore programming aims to increase performance through utilisation of a system's parallel processor cores. In order to assess the effectiveness of our implementation, we investigate its scalability by measuring the performance as we increase the number of available processor cores.

These experiments were performed on a single node in Imperial College's CX1 cluster [7]. Exclusive allocation to the node was requested, and all experiments were performed sequentially on the same machine.

The node featured:

- Two Hex-Core Intel Xeon E5-2620 CPUs @ 2.00GHz, with Hyper-Threading disabled, for twelve physical cores across two sockets. [1] The distribution of cores and memory is detailed further in Appendix A.2.
- 16GB DRAM, split equally between the two sockets.
- Red Hat Enterprise Linux Server 6.4 (Santiago)

KFusion binaries were cross-compiled with gcc 4.8.1 and `-O3` optimisation on 'Hickory' (see Appendix A.3), and glibc-2.19 had to be compiled from source for these binaries and libraries to run.

Attempts were made to compile KFusion with the Intel Compiler Suite [2] for potentially increased performance, but at the time of writing, MARE does not support `icc`, and the MARE version of KFusion fails to compile. The C++ and OpenMP versions compiled successfully with `icc`, however to compare them against a `gcc`-compiled MARE versions would be unfair.

We investigate the performance of the MARE implementation, and compare it to the serial C++ performance, as well as that of the OpenMP implementation, which was later added to the PAMELA KFusion repository.

Because all experiments were run on the same 12 core machine, experiments at lower core counts were restricted to `n` cores with `taskset -c 0-$((n-1))`, which has the effect of restricting the program to the first `n` cores on the machine. The `OMP_NUM_THREADS` environmental variable was used to tell OpenMP how many threads to use, but MARE has no such mechanism in its public API.

Because the cores on the node are split between two sockets, with 0–5 on the first socket, and 6–11 on the second, there will be an expected performance penalty after 6 cores, as KFusion is allocated to memory spanning both NUMA nodes, and memory accesses between sockets is slower.

[1] `http://ark.intel.com/products/64594`
[2] `https://software.intel.com/en-us/intel-compilers/`

All experiments were repeated three times per measurement (i.e. per version and number of cores) and the mean taken. As they were performed with a warm filesystem cache and exclusive access to the node, there was very little variation in performance between these samples.

4.2.1. Wall-Clock Time

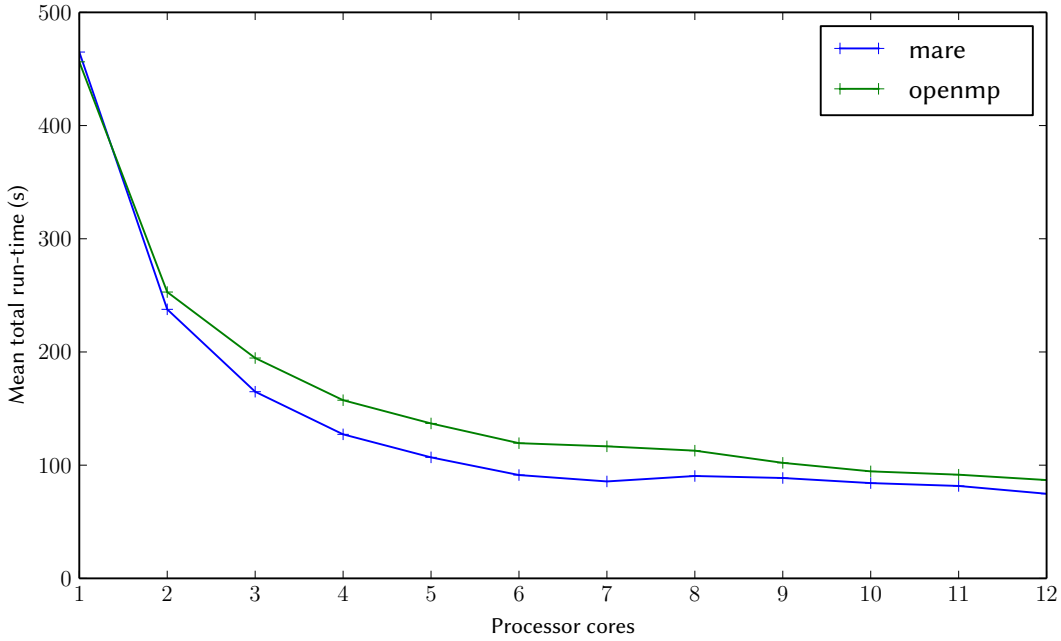
Figure 4.1 shows the mean wall-clock time, and speedup compared to the serial C++ version, of the MARE and OpenMP versions as they process `weird.oni`, a 498 frame recording from the PAMELA dataset. The data can also be found in Table 4.1.

MARE offers an impressive speedup when utilising a small number of cores, Fig. 4.1b shows approximately a $5.0\times$ speedup compared to the serial version when using six cores. Performance degrades when using the cores from the second socket, however, and it takes all 12 cores to attain a $6.1\times$ speedup.

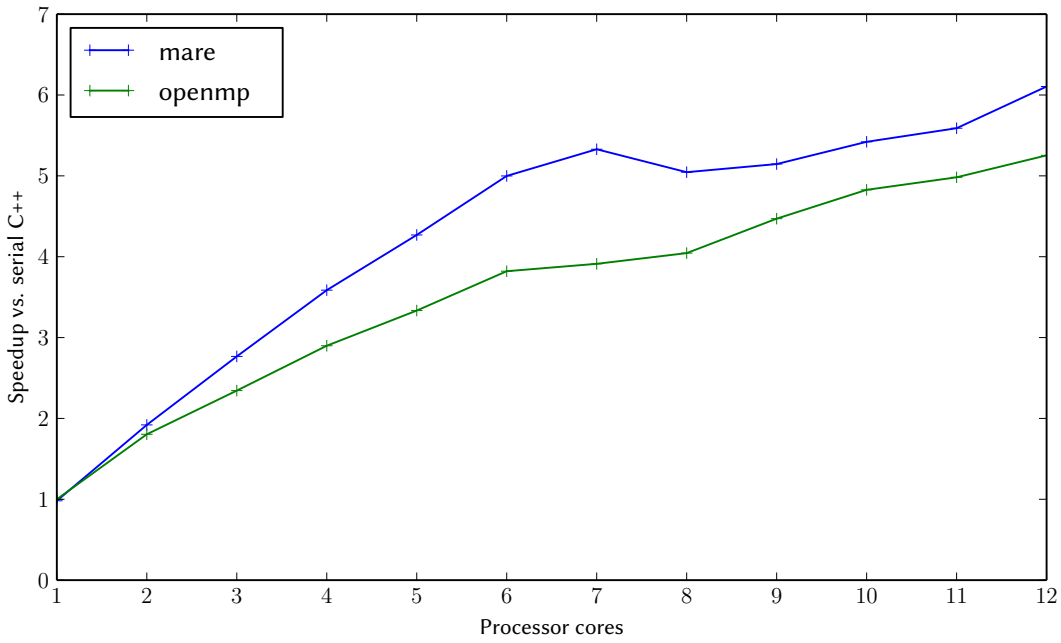
MARE compares favourably to the OpenMP version. With six processor cores, the OpenMP version gains only a $3.8\times$ speedup, and performance degrades more sharply on the second socket, but shows a $5.5\times$ Speedup with all 12 cores.

Cores	MARE Time (s)	MARE Speedup	OpenMP Time (s)	OpenMP Speedup
1	464.940	0.981	456.297	1.000
2	237.663	1.920	252.940	1.804
3	164.897	2.767	194.647	2.344
4	127.227	3.586	157.400	2.899
5	106.887	4.269	136.857	3.334
6	91.277	4.999	119.427	3.821
7	85.623	5.329	116.633	3.912
8	90.437	5.045	112.817	4.044
9	88.670	5.146	102.063	4.471
10	84.170	5.421	94.523	4.827
11	81.637	5.589	91.583	4.982
12	74.740	6.105	86.857	5.253

Table 4.1.: Performance of the MARE and OpenMP versions of KFusion, as the number of cores available increases.



(a) Wall-time performance of the MARE and OpenMP implementations.



(b) Speedup compared to the serial C++ implementation.

Figure 4.1.: Performance of the MARE and OpenMP KFusion implementations as the number of processor cores available varies, on CX1 (see Appendix A.2).

4.2.2. Performance Breakdown

Figure 4.2 and Fig. 4.3 show the stage-by-stage performance of the MARE and OpenMP versions respectively, as the number of cores available increases. The average run-times and speedups of the stages in the MARE version can be found in Table 4.2 and Table 4.3 respectively, and the OpenMP version in Table 4.4 and Table 4.5.

As shown earlier, in Fig. 4.1, MARE is faster than OpenMP throughout.

Note that the 'Acquire' stage refers to the time taken to read a frame with OpenNI2. With the `-f` option to KFusion (see section 3.4), this is essentially constant time, so its lack of speedup is expected. We believe most of this time to be spent in the preparation of the next frame by OpenNI2, and its performance to be limited by single-CPU speed, so scheduling additional work during this time could be an interesting avenue of research.

The 'Preprocess' stage covers a single function call, `KFusion::setKinectDeviceDepth`. Consisting of a single 2D loop, with a short body, the corresponding processing time is very small, and subject to such variance that it does not, in all cases, scale as it would be expected to.

Figure 4.2b shows us that the MARE implementations of raycasting and rendering rendering both scale exceptionally well, with speedups of 8.80 and 8.82 respectively when using all 12 cores. Both take a noticeable efficiency hit when using the second socket, though not until the 8th core.

Integration also scales incredibly well, but suffers more of a slowdown after 7 cores, gaining a $5.4\times$ speedup with 6 cores, but only a $8.2\times$ speedup with 12.

The OpenMP versions of these stages also scale well, and tend to be less affected by the transition to the second socket due to the static scheduling employed by default by GCC's implementation. The performance, however, is usually poorer than that of the MARE implementation (see Fig. 4.3b) – raycasting, rendering and integration net speedups of $8.82\times$, $6.41\times$ and $7.10\times$ respectively.

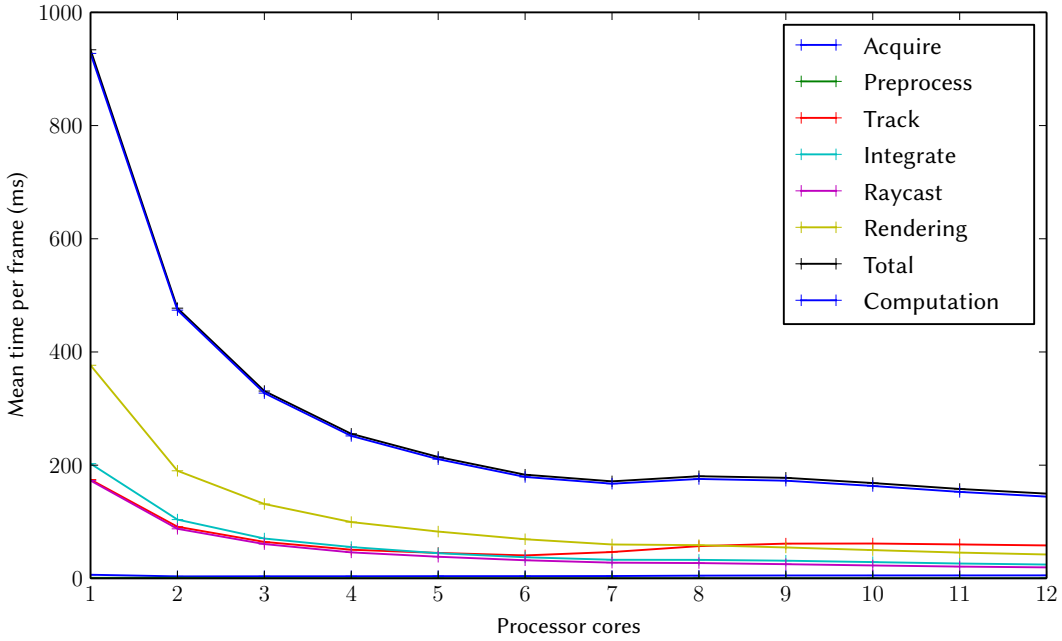
Tracking, however, performs much worse in the face of increasing CPU parallelism, especially across sockets. The MARE implementation sees a speedup of $3.85\times$ with six cores, before degrading to only $2.68\times$ with twelve. Similarly the OpenMP implementation sees a speedup of $3.0\times$ with six cores, but a speedup of $2.73\times$ with twelve.

It is unknown whether this poor speedup is the result of an increase in overhead that comes with the parallelism or an inability to keep the CPUs busy. If it is the latter, then it may be possible to schedule other work to utilise the wasted processor cycles (see chapter 7 for our suggestions for further work in this area).

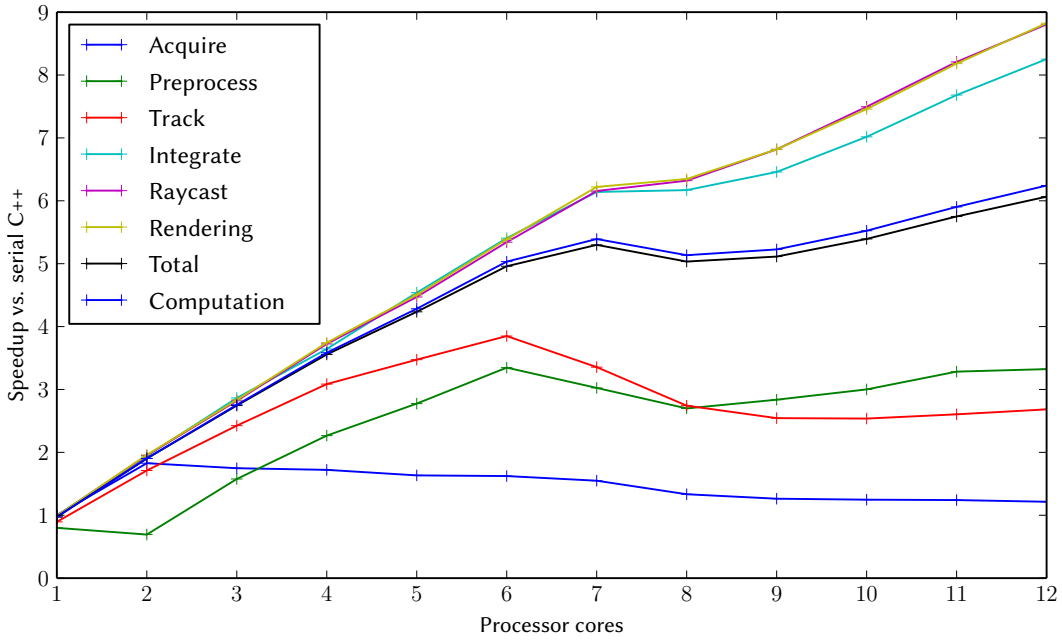
The tracking stage of KFusion consists of many smaller kernels, with implicit barrier synchronisation in between (all parallel work must complete before the algorithm can continue) and a large parallel reduction. Compared to stages such as integration, which consists of one large, three-dimensional loop, the tracking stage is much less optimised for the CPU, and more difficult to port from the CUDA version, which contains many GPU-specific optimisations. It would likely be possible to further optimise the tracking stage for the CPU, minimising memory operations and using MARE to better utilise a CPU parallelism.

We can see, particularly from Fig. 4.2a, that at a low number of cores, rendering and integration dominate the execution time of KFusion. These stages scale well, however, and reduce greatly in execution time as we increase parallelism. With twelve cores, speedup from adding additional

cores has greatly reduced, in large part because tracking now dominates KFusion's execution time, as it does for the GPU versions of the application.

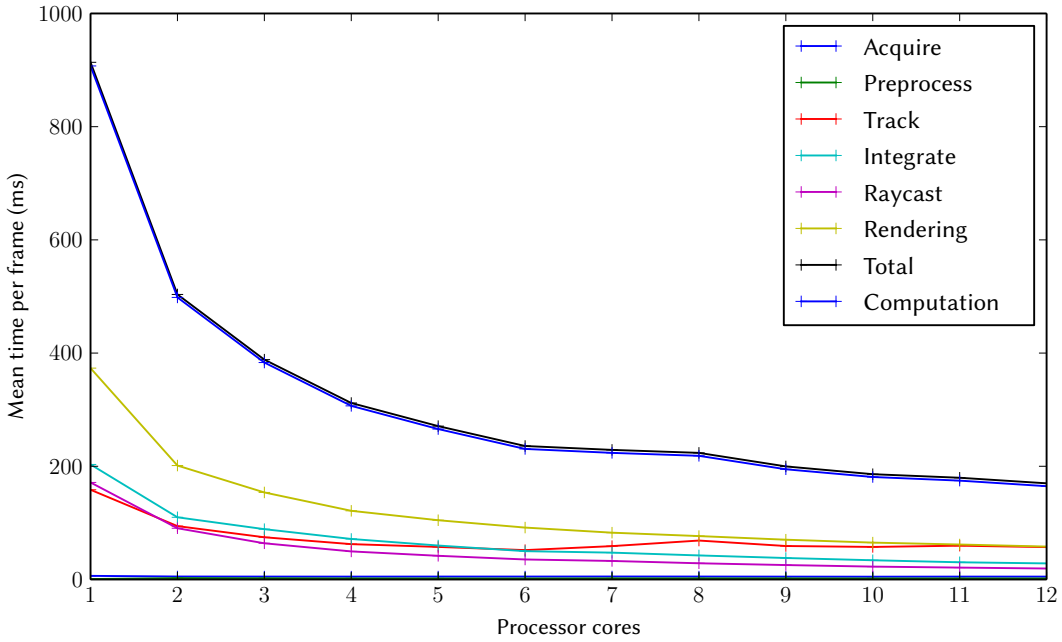


(a) Mean time spent in each stage of the MARE version of KFusion.

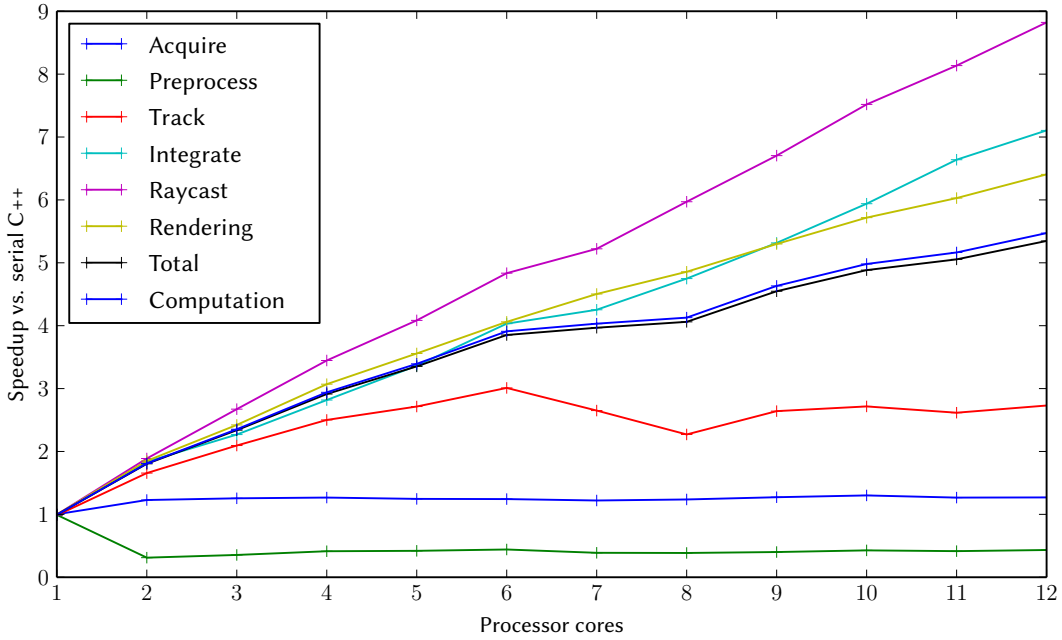


(b) Speedup of each stage of the MARE version of KFusion, compared to the serial C++ implementation.

Figure 4.2.: Performance of the MARE implementation of KFusion, as the number of cores available varies, on CX1 (see Appendix A.2).



(a) Mean time spent in each stage of the OpenMP version of KFusion.



(b) Speedup of each section of the OpenMP version of KFusion, compared to the serial C++ implementation.

Figure 4.3.: Performance of the OpenMP implementation of KFusion, as the number of cores available varies, on CX1 (see Appendix A.2).

Cores	Acquire	Preprocess	Track	Integrate	Raycast	Rendering	Total	Computation
1	6.397	0.934	174.551	202.720	172.783	376.505	933.890	927.493
2	3.508	1.078	91.255	103.877	87.571	190.098	477.386	473.878
3	3.668	0.474	64.412	70.411	60.595	131.419	330.978	327.310
4	3.724	0.330	50.621	55.350	45.892	99.470	255.386	251.662
5	3.924	0.269	44.961	44.434	38.225	82.684	214.497	210.573
6	3.950	0.223	40.575	37.309	31.991	69.119	183.167	179.217
7	4.138	0.247	46.550	32.847	27.761	59.871	171.415	167.276
8	4.806	0.277	56.951	32.693	27.042	58.684	180.452	175.646
9	5.076	0.263	61.389	31.231	25.065	54.625	177.650	172.573
10	5.139	0.249	61.551	28.745	22.800	49.937	168.420	163.282
11	5.165	0.227	59.961	26.259	20.818	45.522	157.953	152.788
12	5.279	0.225	58.199	24.440	19.409	42.207	149.759	144.480

Table 4.2.: Mean wall-clock time (in ms) of each stage of the MARE version of KFusion, as the number of cores available increases.

Cores	Acquire	Preprocess	Track	Integrate	Raycast	Rendering	Total	Computation
1	1.002	0.800	0.895	0.995	0.989	0.989	0.973	0.972
2	1.828	0.693	1.712	1.942	1.952	1.959	1.903	1.903
3	1.748	1.576	2.425	2.865	2.820	2.834	2.745	2.756
4	1.722	2.266	3.085	3.644	3.724	3.744	3.557	3.584
5	1.634	2.775	3.474	4.539	4.471	4.504	4.235	4.283
6	1.623	3.347	3.849	5.406	5.342	5.388	4.959	5.033
7	1.549	3.025	3.355	6.141	6.156	6.221	5.299	5.392
8	1.334	2.699	2.742	6.170	6.320	6.347	5.034	5.135
9	1.263	2.837	2.544	6.459	6.818	6.818	5.113	5.227
10	1.248	3.001	2.538	7.017	7.496	7.458	5.394	5.524
11	1.241	3.284	2.605	7.681	8.209	8.182	5.751	5.903
12	1.215	3.324	2.684	8.253	8.805	8.824	6.066	6.243

Table 4.3.: Mean speedup of each stage of the MARE version of KFusion, relative to the serial C++ version, as the number of cores available increases.

Cores	Acquire	Preprocess	Track	Integrate	Raycast	Rendering	Total	Computation
1	6.400	0.753	158.505	203.214	171.429	373.412	913.713	907.313
2	5.221	2.398	94.293	109.863	90.504	201.299	503.577	498.357
3	5.108	2.107	74.569	88.906	63.937	153.772	388.398	383.290
4	5.060	1.805	62.441	71.579	49.585	121.320	311.789	306.729
5	5.146	1.778	57.535	59.970	41.842	104.670	270.940	265.794
6	5.158	1.693	51.881	50.023	35.346	91.711	235.813	230.655
7	5.253	1.922	58.927	47.424	32.724	82.693	228.944	223.690
8	5.184	1.935	68.789	42.473	28.621	76.676	223.679	218.495
9	5.039	1.863	59.127	37.940	25.491	70.285	199.745	194.706
10	4.928	1.748	57.502	33.959	22.731	65.133	186.002	181.074
11	5.062	1.799	59.696	30.381	21.006	61.774	179.719	174.657
12	5.051	1.724	57.199	28.389	19.371	58.140	169.874	164.823

Table 4.4.: Mean wall-clock time (in ms) of each stage of the OpenMP version of KFusion, as the number of cores available increases.

Cores	Acquire	Preprocess	Track	Integrate	Raycast	Rendering	Total	Computation
1	1.002	0.992	0.985	0.993	0.997	0.997	0.994	0.994
2	1.228	0.312	1.656	1.836	1.888	1.850	1.804	1.810
3	1.255	0.355	2.095	2.269	2.673	2.422	2.339	2.353
4	1.267	0.414	2.501	2.818	3.447	3.070	2.913	2.941
5	1.246	0.420	2.715	3.363	4.084	3.558	3.353	3.394
6	1.243	0.441	3.010	4.032	4.835	4.061	3.852	3.911
7	1.220	0.389	2.650	4.253	5.223	4.504	3.968	4.032
8	1.237	0.386	2.271	4.749	5.971	4.857	4.061	4.128
9	1.272	0.401	2.642	5.316	6.704	5.299	4.548	4.633
10	1.301	0.427	2.716	5.940	7.518	5.718	4.884	4.981
11	1.267	0.415	2.616	6.639	8.136	6.029	5.055	5.164
12	1.269	0.433	2.731	7.105	8.823	6.406	5.347	5.472

Table 4.5.: Mean speedup of each stage of the OpenMP version of KFusion, relative to the serial C++ version, as the number of cores available increases.

4.2.3. Energy Consumption

We were curious how the MARE implementation compared in terms of energy usage, to both the serial C++ version, and to the GPU-accelerated versions.

We tested each implementation processing `weird.oni` on 'Vera', a commodity desktop machine featuring:

- An Intel Core i5-2500K CPU. [3] Quad-Core, no Hyper-Threading. Clock Speed: 3.30GHz, Max. Turbo Frequency: 3.70GHz.
- 16GB 1600MHz DDR3 RAM (4 × Corsair CMZ8GX3M2A1600C9).
- An MSI GeForce GTX 670 'Power Edition' (NVIDIA GK104, 2GB GDDR5 RAM). [4,5]
- An ASUS P8Z68-V LX Motherboard.
- Arch Linux x86_64, package versions:
  - linux 3.14.6-1
  - gcc 4.9.0-4
  - openni2 2.2beta2-3
  - nvidia 337.25-1
  - opencl-nvidia 337.25-1
  - cuda 6.0.37-2
  - libcl 1.1-3
  - opencl-headers12 1:1.2.r26813-1

Further information on the arrangement of memory, CPU cores and cache can be found in Appendix A.1.

Each run was repeated three times, and power measurements were taken at regular intervals with an Olson RMB/C13/C14 inline remote power monitoring meter, and the mean taken. Note that this will likely be an under-approximation of the power used while processing, as each implementation spends some time waiting for the next frame to be processed, where it will be using a single CPU core. [6] By taking the mean power usage, subtracting the power used when idle, and multiplying by the mean run-time, we were able to calculate the energy used by each implementation to process the test file, and the results are shown in Table 4.6.

We can see with the pure CPU implementations (CPP, MARE and OpenMP) that the faster an implementation, the greater its power draw, but the lower the total energy usage. I.e. the MARE implementation uses $5,130.1J$ over $89.5s$, whereas the OpenMP implementation uses $5178.0J$ over $118.5s$, making maximum utilisation of the CPU the better strategy for reducing

[3] `http://ark.intel.com/products/52210`

[4] `http://www.msi.com/product/vga/N670_PE_2GD5OC.html`

[5] `http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-670/specifications`

[6] A `http://www.olson.co.uk/rm_box.htm`, with IEC 60320 C13/C14 connectors.

Version	Mean Power (W)	Power - Idle (W)	Mean Time (s)	Mean Energy (J)
cpp	92.250	21.250	314.817	6689.854
mare	128.294	57.294	89.540	5130.115
openmp	114.688	43.688	118.523	5177.988
cuda	128.000	57.000	9.343	532.570
opencl	163.680	92.680	74.170	6874.076
optopencl	122.833	51.833	11.783	610.769

Table 4.6.: Power and energy usage of each version of KFusion, while testing `weird.oni` on 'Vera'. Energy usage was calculated from the Power usage minus the idle power usage of $71W$.

total power usage, when faced with a fixed-length input (the MARE implementation uses an average 382% CPU, and the OpenMP an average 325%).

From this experiment it is clear, however, that without further optimisation, the CPU-based implementations cannot compete for efficiency with either GPU implementation, on a system with a powerful GPU, such as Vera's. Note that at the time of writing there are two OpenCL builds available in PAMELA's KFusion fork, 'opencl' and 'optopencl', which has additional optimisations. Both the CUDA and optimised OpenCL ('optopencl') implementations run almost ten times faster than even the MARE version, with similar power draws.

4.3. Summary

We presented our data parallel MARE implementation of KFusion, and found it to outperform the OpenMP implementation in almost all cases, as well as offering increased energy efficiency due to the shorter run-time.

We have shown that raycasting and rendering performance scale exceedingly well with the addition of CPU cores, even across socket boundaries, while integration also scales very well, but is noticeably affected by the socket transition.

We have found, however, that as the number of cores available increases, the execution of KFusion becomes dominated by the tracking stage, which scales poorly, particularly across sockets. Without a further optimised tracking stage, KFusion on the CPU will be at a severe disadvantage, in performance and energy efficiency, compared to the GPU implementations.

5. Evaluation

In our performance evaluation, section 4.2, we have shown that our MARE KFusion implementation performs more efficiently than the OpenMP version, by better exploiting the underlying parallelism. We showed that it scales well with increasing parallelism, making good use of available compute resources, with a $5\times$ speedup for six processor cores.

Performance is still an order of magnitude below that of the GPU implementations. Beyond approximately six cores, performance was held back by poor tracking scalability, the key bottleneck in our experiments. We show in section 4.2.2 that tracking performance fails to scale as effectively as other stages of the algorithm, and give possible explanations for this. In chapter 7 we present a number of possible research directions to better optimise it for the CPU.

With our analysis of the data flow in chapter 3, we identified key stages of KFusion which could, with modifications, be executed asynchronously. This may help to compensate for poor tracking scalability, or could effectively utilise otherwise wasted cycles while KFusion waits for frame acquisition. See chapter 7 for a thorough explanation of these potential avenues of research.

5.1. Evaluation of MARE

MARE's performance, as shown in section 4.2, is excellent. Likely due to being designed for use on a wide variety of mobile devices under varying load, MARE's low-overhead dynamic scheduling proved adept at exploiting the parallelism of the underlying hardware

The data parallel 'patterns' that MARE provides proved easy to use. They did, however, require a greater number of code changes than OpenMP's `#pragma omp parallel for`, and lack the ability to declare variables as `private`, instead requiring their declarations to be moved within the loop (though this is a more concise and readable solution).

One of OpenMP's most compelling features is that when the code is compiled with OpenMP support disabled, or with a compiler where such support never existed, the directives are treated as comments and ignored, and what remains is a functional serial program. MARE provides no such mechanism (though a serial implementation of its API would be possible). MARE also lacks the ability to specify the number of threads to use at runtime (à la OpenMP's `OMP_NUM_THREADS`), but this proved not to be too much of a problem, as its dynamic scheduler responded well to being limited with `taskset` (see section 4.2).

MARE's task parallelism constructs, while powerful, proved tricky to apply to a mature, data-heavy application like KFusion without rewriting much of the code, but may be well suited to new applications being developed from scratch.

Porting an existing application to use MARE tasks, or even the SDF API is a time-consuming process, which may not even yield significant performance benefits. We expect that qualitative

analysis of the program can reveal likely sources of parallelism that can be exploited directly with significantly reduced effort.

5.2. Experience Working with Ported CUDA Code

KFusion's C++ version was adapted from the CUDA implementation, so holds a number of interesting characteristics:

- Many loops were present with large degrees of parallelism and independent body calculations. These remnants of CUDA kernel invocations proved extremely useful when exploiting the same data parallelism on the CPU.
- In order to better utilise the underlying hardware, many CUDA kernels iterated through 2D images in the opposite order to that in which they are laid out in memory. While appropriate for GPU hardware, these orderings were counter-productive for CPU code, and had to be reversed – though performance gained was surprisingly small, but not insignificant.
- Some code, particularly the multi-stage parallel reduction present in the tracking stage, is heavily optimised for CUDA execution, where data is accumulated by successively fewer threads copying data between memory levels. With one, single memory space, this is entirely unnecessary on the CPU, but disentangling the CUDA code to simplify the algorithm proved difficult.

6. Conclusion

In section 4.2.2 we found that much of KFusion is highly scalable on the CPU, and our MARE implementation is adept at exploiting many processor cores. Overall performance is limited, however, by the poor scalability of the tracking stage, which becomes a significant part of the processing time for large numbers of parallel cores. We suggest a number of potential solutions to better optimise the tracking stage for the CPU.

In our analysis in chapter 3, we investigated the difficulties of exploiting task-parallelism in a data-heavy, interactive application like KFusion. We outlined a number of promising areas for future research in this area, along with the challenges involved.

6.1. Summary

As summarised in section 1.4, the major contributions in this work were as follows.

We created, in collaboration with Luigi Nardi and other PAMELA project members, a new off-line dense SLAM performance benchmark based upon KFusion. This benchmark proved to be reliable, with results easy to reproduce, and flexible in the systems and technologies upon which it could be run, while providing valuable insight into the performance of KFusion on a multitude of systems.

We created an efficient multicore implementation of KFusion using Qualcomm MARE. This implementation proved to be highly efficient compared to the existing CPU implementations, and provided valuable insight into the performance bottlenecks of KFusion. We evaluated both the usability and performance of MARE in this use case, and found it to be easy to use and highly performant in a wide variety of situations.

Investigating the scalability of our implementation, we discovered precisely the performance bottlenecks in KFusion, with our utilisation of multicore hardware limited by the poor scalability of the tracking and acquisition stages.

We performed a quantitative and qualitative analysis of the performance and data flow structure of KFusion, showing the potential for both task and data parallelism. We provide insight into likely targets for future work exploiting the task parallelism in KFusion, and in similar computer vision applications, for performance and energy efficiency.

7. Further Work

The analysis presented in this work suggests that there is promising scope for exploring parallelism within KFusion.

Tracking Optimisation By better optimising the Track stage for the CPU, it should be possible to increase the scalability of KFusion as a whole for multicore hardware. This could be done by undoing the CUDA-esque optimisations that remain, removing superfluous data copying, and making use of MARE primitives for reduction, rather than the multi-stage reduction developed for CUDA.

The tracking phase consists of many small kernels, with intermediate buffers and barrier synchronisations in between, making it a prime candidate for loop fusion techniques, if at all possible, to reduce the parallelism overheads involved.

Dynamic Integration Rate When KFusion has built an accurate map of the environment, and tracking is performing accurately against this map, successive integration operations may not always be valuable. The integration rate could be varied dynamically, or shut off completely when the map is built. This would save power when operating in small, static environments, and still permit high-accuracy tracking.

It could also be possible to detect, on a frame-by-frame basis, the time taken by the tracking stage, and adjust integration accordingly. For example, during sharp camera movements, the time taken by the tracking phase may peak, before quickly reducing again – it may be advantageous to delay integration after a log-running track operation, instead performing it on a 'quiet frame'. This would help to maintain a more consistent frame-rate, while making better use of available processing time.

Asynchronous integration By buffering the input to the integration step, it should be possible to perform it asynchronously, while tracking continues. This would allow MARE to keep the CPU fully utilised between invocations of other parallel kernels, and in particular during the Acquire and Track stages, compensating for their poor scalability by performing other work at the same time. This would come at the expense of memory locality, which could possibly be resolved by pinning tasks to particular CPU cores, but MARE does not currently expose such an API.

The raycasting and tracking stages should be tolerant of slightly stale and partially-updated data, so this should be perfectly safe. There will be a slight reduction in accuracy, but it should be within tolerable limits. It will, however, greatly reduce the computation that is required before KFusion can begin to process the next frame, allowing it to more smoothly meet real-time deadlines.

Asynchronous Preprocessing, Rendering and Raycasting It should be possible to begin the preprocessing of a frame, including the preparatory first stages of tracking asynchronously, before the raycasting of the previous frame has completed.

Similarly, after integration has completed, much of the raycasting and rendering stages could be executed simultaneously, and only raycasting has to complete before the next iteration of tracking can commence (rendering is required for the UI only).

By reducing the amount of processing required in the critical tracking loop, increasing the available parallelism, it should be possible to increase performance and flexibility, even while operating in a continuous, real-time environment.

As with asynchronous integration, this asynchrony could allow MARE to keep the processor fully utilised between stages, and during frame acquisition and tracking, but the performance hit that would result from the loss of locality is unknown.

Synchronous Data Flow There is potential to use the MARE SDF API to automatically manage task parallelism and data-flow and pipeline the computation, by explicitly modelling the dependence graph. While the SDF API is designed for channelling small amounts of data, copied by value between nodes, it should be possible to transparently create and reuse intermediate buffers for the large amounts of data passed in KFusion. This approach would require a great deal of work, but has the potential to uncover interesting parallelism in KFusion, and to fully utilise the CPU at all times.

Heterogeneous Computing As introduced in section 3.3.4, it would be possible to use MARE's Heterogeneous Compute API to divide tasks between the CPU and the GPU, with MARE automatically synchronising data between them when required.

Such a split could be to perform tracking on one device, and integration, raycasting and rendering on the other, as the tracking stage does not require the integration volume. The problem with this split would be that tracking performance on the CPU is quite poor, as shown in section 4.2.2, though an implementation better optimised for the CPU could be more efficient.

With many CPU cores, and asynchronous integration, performing integration, raycasting and rendering on the CPU, and tracking on the GPU could also be interesting.

For future pipelines, developed from scratch, techniques like these could be employed from an early stage, greatly reducing the cost of adoption, but bringing increased flexibility and performance.

8. Bibliography

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 'Cilk: An Efficient Multithreaded Runtime System'. In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, 1995, pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958.

[2] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 'hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications'. English. In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italy, Feb. 2010. DOI: 10.1109/PDP.2010.67. URL: http://hal.inria.fr/inria-00429889.

[3] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. 'MonoSLAM: Real-Time Single Camera SLAM'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.6 (2007), pp. 1052–1067.

[4] *Grand Central Dispatch (GCD) Reference – Mac Devleoper Library*. URL: https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html (visited on 2014-01-30).

[5] Ankur Handa, Richard A. Newcombe, Adrien Angeli, and Andrew J. Davison. 'Real-Time Camera Tracking: When is High Frame-rate Best?' In: *Proceedings of the 12th European Conference on Computer Vision - Volume Part VII*. ECCV'12. Florence, Italy: Springer-Verlag, 2012, pp. 222–235. ISBN: 978-3-642-33785-7. DOI: 10.1007/978-3-642-33786-4_17.

[6] W. Daniel Hillis and Guy L. Steele Jr. 'Data Parallel Algorithms'. In: *Commun. ACM* 29.12 (Dec. 1986), pp. 1170–1183. ISSN: 0001-0782. DOI: 10.1145/7902.7903.

[7] *Imperial College High Performance Computing Service*. URL: https://www.imperial.ac.uk/ict/services/teachingandresearchservices/highperformancecomputing (visited on 2014-06-07).

[8] Qualcomm Technologies Inc. *Multicore Asynchronous Runtime Environment – Documentation and Interface Specification*. Apr. 28, 2014. URL: https://developer.qualcomm.com/mobile-development/maximize-hardware/harnessing-multicore-socs-mare/parallel-computing-mare-tools-and-resources (visited on 2014-05-14).

[9] Qualcomm Technologies Inc. *Qualcomm® MARE – Enabling Applications for Heterogeneous Mobile Devices*. Apr. 25, 2014. URL: https://developer.qualcomm.com/mobile-development/maximize-hardware/harnessing-multicore-socs-mare/parallel-computing-mare-tools-and-resources (visited on 2014-05-14).

[10] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. 'KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera'. In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. 2011, pp. 559–568. URL: https://research.microsoft.com/apps/pubs/default.aspx?id=155416.

[11] Samir Kumar. 'Exploiting Multicore Performance for Your Native Apps'. In: *UPLINQ* (Sept. 4, 2013). URL: http://files.meetup.com/1778543/2013_10_07_Exploiting-Multicore-Performance-Native-Apps_LR.pdf (visited on 2014-06-12).

[12] Richard A Newcombe, Andrew J Davison, Shahram Izadi, Pushmeet Kohli, Otmar Hilliges, Jamie Shotton, David Molyneaux, Steve Hodges, David Kim, and Andrew Fitzgibbon. 'KinectFusion: Real-time dense surface mapping and tracking'. In: *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*. IEEE. 2011, pp. 127–136. URL: https://research.microsoft.com/apps/pubs/default.aspx?id=155378.

[13] *OpenMP Application Program Interface Version 4.0*. July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (visited on 2014-01-30).

[14] David A Patterson and John L Hennessy. *Computer Organization And Design: The Hardware/Software Interface*. Revised Fourth Edition. Morgan Kaufmann, 2011. ISBN: 978-0-12-374750-1.

[15] Alex Prokopec and Dmitry Petrashko. 'ScalaBlitz Efficient Collections Framework'. Scala eXchange 2013. Nov. 2, 2013. URL: https://skillsmatter.com/skillscasts/4467-macro-based-scala-parallel-collections (visited on 2014-01-30).

[16] Gerhard Reitmayr. *GerhardR/kfusion · GitHub*. URL: https://github.com/GerhardR/kfusion (visited on 2014-01-30).

[17] Renato F Salas-Moreno, Richard A Newcombe, Hauke Strasdat, Paul HJ Kelly, and Andrew J Davison. 'Slam++: Simultaneous localisation and mapping at the level of objects'. In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, pp. 1352–1359.

[18] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.

Appendices

A. System Specifications

We include here additional information on any systems used for testing, including hardware locality diagrams created by `lstopo` [2].

A.1. 'Vera'

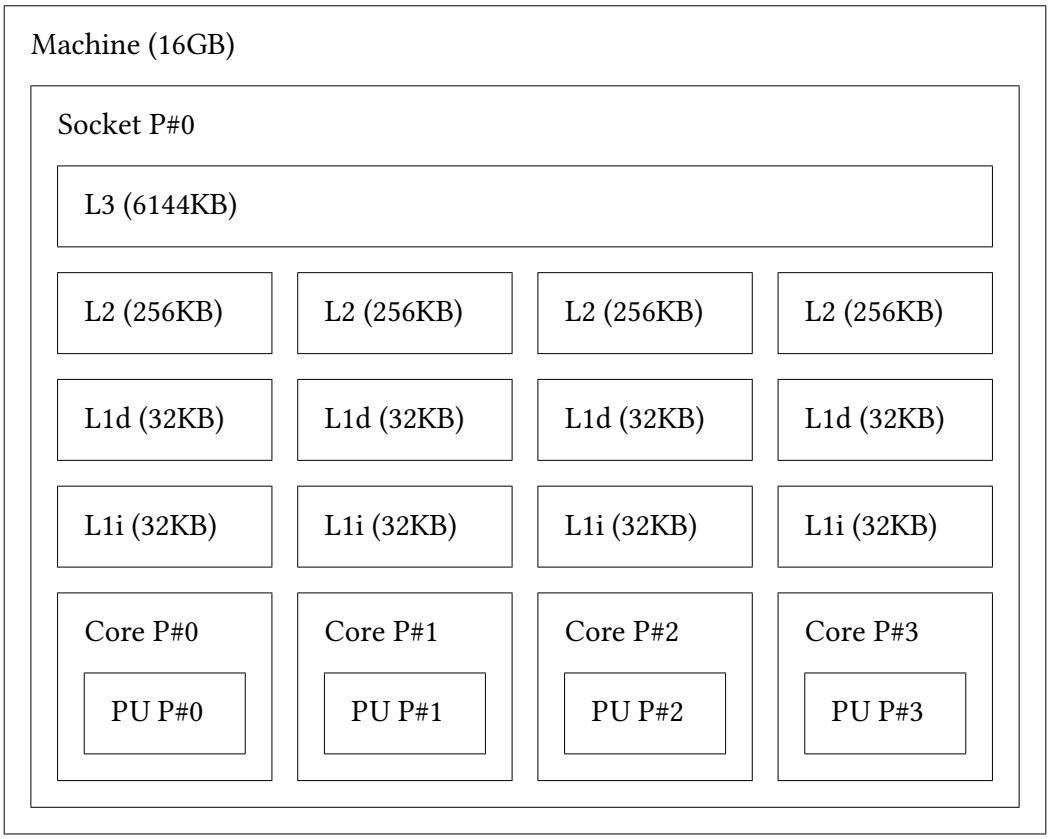


Figure A.1.: The arrangement of CPU cores and cache on `Vera`, as reported by `lstopo`.

A.2. CX1

Figure A.2.: The CPU cores and cache on the CX1 node, as reported by `lstopo`.

A.3. ‘Hickory’

- Intel Core i7-4770K CPU. [1] Quad-Core, 8 threads with Hyper-Threading. Clock Speed: 3.5 GHz, Max. Turbo Frequency: 3.9 GHz.
- NVIDIA GeForce GTX Titan (NVIDIA GK110). [2]
- Ubuntu 13.04 (Raring Ringtail) x86_64, packages:
  - linux 3.8.0-35-generic
  - gcc 4.8.1-2ubuntu1~13.04
  - OpenNI2-2.2-beta2, compiled from source.
  - nvidia-331 331.20-0ubuntu1~xedgers~raring1

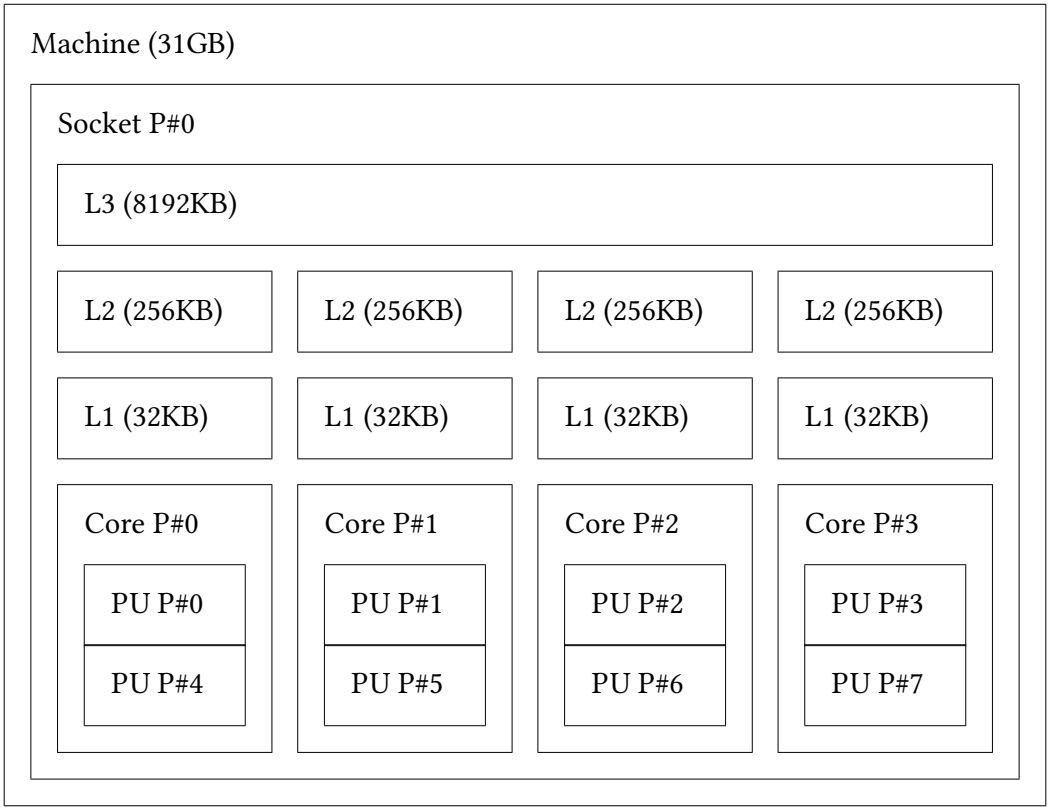


Figure A.3.: The arrangement of CPU cores and cache on `Hickory`, as reported by `lstopo`.

[1] `http://ark.intel.com/products/75123`
[2] `http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-titan/specifications`