# Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM

Luigi Nardi[1], Bruno Bodin[2], M. Zeeshan Zia[1], John Mawer[3], Andy Nisbet[3], Paul H. J. Kelly[1],
Andrew J. Davison[1], Mikel Luján[3], Michael F. P. O'Boyle[2], Graham Riley[3], Nigel Topham[2] and Steve Furber[3]
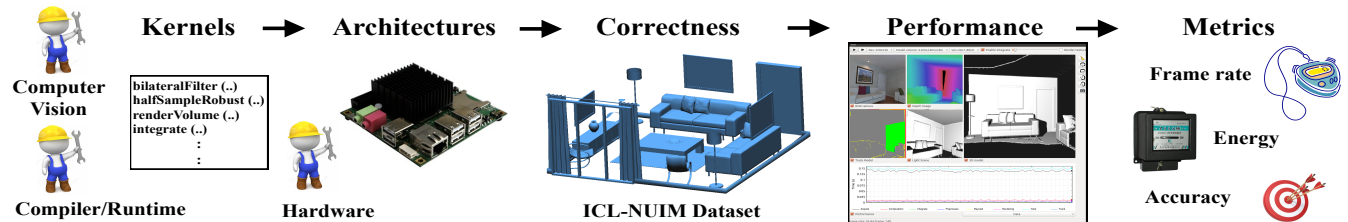
Fig. 1: The SLAMBench framework makes it possible for experts coming from the computer vision, compiler, run-time, and hardware communities to cooperate in a unified way to tackle algorithmic and implementation alternatives.

*Abstract*— Real-time dense computer vision and SLAM offer great potential for a new level of scene modelling, tracking and real environmental interaction for many types of robot, but their high computational requirements mean that use on mass market embedded platforms is challenging. Meanwhile, trends in low-cost, low-power processing are towards massive parallelism and heterogeneity, making it difficult for robotics and vision researchers to implement their algorithms in a performance-portable way. In this paper we introduce SLAMBench, a publicly-available software framework which represents a starting point for quantitative, comparable and validatable experimental research to investigate trade-offs in performance, accuracy and energy consumption of a dense RGB-D SLAM system. SLAMBench provides a KinectFusion implementation in C++, OpenMP, OpenCL and CUDA, and harnesses the ICL-NUIM dataset of synthetic RGB-D sequences with trajectory and scene ground truth for reliable accuracy comparison of different implementation and algorithms. We present an analysis and breakdown of the constituent algorithmic elements of KinectFusion, and experimentally investigate their execution time on a variety of multicore and GPU-accelerated platforms. For a popular embedded platform, we also present an analysis of energy efficiency for different configuration alternatives.

## I. INTRODUCTION

Recent advances in computer vision have led to real-time algorithms for dense 3D scene understanding, such as KinectFusion [19], which estimate the pose of a depth camera while building a highly detailed 3D model of the environment. Such real-time 3D scene understanding capabilities can radically change the way robots interact with and manipulate the world.

[1]Dept. Computing, Imperial College, London, UK {luigi.nardi, zeeshan.zia, p.kelly, ajd}@imperial.ac.uk

[2]Institute for Computing Systems Architecture, The Univ. of Edinburgh, Scotland {bbodin, mob, npt}@inf.ed.ac.uk

[3]School of Computer Science, Univ. of Manchester, UK {andy.nisbet, john.mawer, mikel.lujan, graham.riley, steve.furber}@manchester.ac.uk

While classical point feature-based *simultaneous localisation and mapping* (SLAM) techniques are now crossing into mainstream products via embedded implementation in projects like Project Tango [3] and Dyson 360 Eye [1], dense SLAM algorithms with their high computational requirements are largely at the prototype stage on PC or laptop platforms. Meanwhile, there has been a great focus in computer vision on developing benchmarks for accuracy comparison, but not on analysing and characterising the envelopes for performance and energy consumption.

The growing interest in real-time applications means that the difficulty of parallelisation affects every application developer. Existing benchmarks are typically general purpose, e.g. the MediaBench suite [16] for evaluation of embedded multimedia and communications, the DaCapo benchmark [7] for sequential client and server Java code, and the SPEC CPU benchmark [14] which stresses the processor, memory subsystem and compiler. Such benchmarks have proved valuable for the evaluation of existing scientific and engineering applications, but they fail to capture essential elements of modern computer vision algorithms and, in particular, SLAM applications. To address these issues, we present SLAMBench [4] (see Fig. 1) which is a portable implementation of KinectFusion with verifiable accuracy relative to a known ground truth. SLAMBench allows computer vision researchers to provide and evaluate alternative algorithmic implementations but it also enables domain-specific optimisation, auto-tuning and domain-specific languages (DSLs) in dense SLAM. It takes a step forward improving the way academia and industry evaluate system design and implementation in computer vision. To the best of our knowledge, SLAMBench is the first performance, energy and accuracy benchmark dedicated to 3D scene understanding applications.

The contributions of this paper are:

- We present a SLAM performance benchmark that combines a framework for quantifying quality-of-result with instrumentation of execution time and energy consumption.
- We present a qualitative analysis of the parallel patterns underlying each of the constituent computational elements of the KinectFusion algorithm.
- We present a characterisation of the performance of the KinectFusion implementation using both multicore CPUs and GPUs, on a range of hardware platforms from desktop to embedded.
- We present an analysis of the algorithm's energy consumption under various implementation configurations using an embedded, multicore system on chip (SoC) with GPU acceleration.
- We offer a platform for a broad spectrum of future research in jointly exploring the design space of algorithmic and implementation-level optimisations.

## II. RELATED WORK

Computer vision research has traditionally focused on optimising the accuracy of algorithms. In autonomous driving, for example, the KITTI benchmark suite [11] provides data and evaluation criteria for the stereo, optical flow, visual odometry and 3D object recognition. The ICL-NUIM dataset [12] and TUM RGB-D benchmark [23] aim to benchmark the accuracy of visual odometry and SLAM algorithms. However, in an energy and performance constrained context, such as a battery-powered robot, it is important to achieve sufficient accuracy while maximising battery life. New benchmarks are needed which provide tools and techniques to investigate these constraints. An important early benchmark suite for performance evaluation entirely dedicated to computer vision is SD-VBS [25]. SD-VBS provides single-threaded C and MATLAB implementations of 28 commonly used computer vision kernels that are combined to build 9 high-level vision applications; only some modules relevant to SLAM are included, notably Monte Carlo localisation. SD-VBS [25] prohibits modifications to the algorithm, only allowing the implementation to be tuned to suit novel hardware architectures. This limits the use of the benchmark in the development of novel algorithms.

Another attempt at such performance evaluation is MEVBench [9], which focuses on a set of visual recognition applications including face detection, feature classification, object tracking and feature extraction. It provides single and multithreaded C++ implementations for some of the kernels with a special emphasis on low-power embedded systems. However, MEVBench only focuses on recognition algorithms and does not include a SLAM pipeline.

While such efforts are a step in the right direction, they do not provide the software tools for accuracy verification and exploitation of hardware accelerators or *graphics processor units* (GPUs). Nor do they enable investigation of energy consumption, performance and accuracy envelopes for 3D scene reconstruction algorithms across a range of hardware targets. The lack of benchmarks stems from the difficulty in

systematically comparing the accuracy of the reconstruction while measuring the performance. In this work we focus specifically on SLAM and introduce a publicly-available framework for quantitative, comparable and validatable experimental research in the form of a benchmark for dense 3D scene understanding. A key feature of SLAMBench is that it is designed on top of the recently-proposed ICL-NUIM accuracy benchmark [12], and thus supports wider research in hardware and software. The quantitative evaluation of solution accuracy into SLAMBench enables algorithmic research to be performed. The latter is an important feature that is lacking in current performance benchmarks. A typical output of SLAMBench consists of the performance achieved, and the accuracy of the result along with the energy consumption (on platforms where such measurement is possible). These parameters capture the potential trade-offs for real-time vision platforms.

## III. SLAM AND KINECTFUSION

SLAM systems aim to perform real-time localisation and mapping "simultaneously" for a sensor moving through an unknown environment. SLAM could be part of a mobile robot, enabling the robot to update its estimated position in an environment, or an augmented reality (AR) system where a camera is moving through the environment, allowing the system to render graphics at appropriate locations in the scene (e.g. to illustrate repair procedures in-situ [13] or to animate a walking cartoon character on top of a table). Localisation typically estimates the location and pose of the sensor (e.g. a camera) with regard to a map which is extended as the sensor explores the environment.

The KinectFusion [19] algorithm utilises a depth camera to perform real-time localisation and dense mapping. A single raw depth frame from these devices has both noise and holes. KinectFusion registers and fuses the stream of measured depth frame obtained as the scene is viewed from different viewpoints into a clean 3D geometric map. While it is beyond the scope of this paper to go into the details of the KinectFusion algorithm, we briefly outline the key computational steps involved in the top row of Fig. 2. KinectFusion normalizes each incoming depth frame and applies a bilateral filter [24] (*Preprocess*); before computing a point cloud (with normals) for each pixel in the camera frame of reference. Next, KinectFusion estimates (Track) the new 3D pose of the moving camera by registering this point cloud with the current global map using a variant of *iterative closest point* (ICP) [6]. Once the new camera pose has been estimated, the corresponding depth map is fused into the current 3D reconstruction (*Integrate*). KinectFusion utilises a voxel grid as the data structure to represent the map, employing a *truncated signed distance function* (TSDF) [10] to represent 3D surfaces. The 3D surfaces are present at the zero crossings of the TSDF and can be recovered by a raycasting step, which is also useful for visualising the reconstruction. The key advantage of the TSDF representation is that it simplifies fusion of new data into existing data to
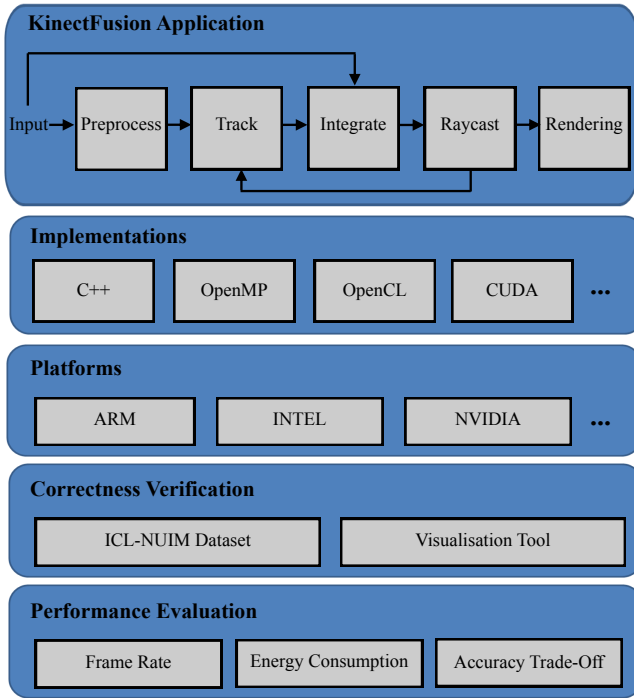
Fig. 2: SLAMBench framework.

| Kernels | Pipeline | Pattern | In | Out | % |
|---|---|---|---|---|---|
| acquire | Acquire | n/a | pointer | 2D | 0.03 |
| mm2meters | Preprocess | Gather | 2D | 2D | 0.06 |
| bilateralFilter | Preprocess | Stencil | 2D | 2D | 33.68 |
| halfSample | Track | Stencil | 2D | 2D | 0.05 |
| depth2vertex | Track | Map | 2D | 2D | 0.11 |
| vertex2normal | Track | Stencil | 2D | 2D | 0.27 |
| track | Track | Map/Gather | 2D | 2D | 4.72 |
| reduce | Track | Reduction | 2D | 6x6 | 2.99 |
| solve | Track | Sequential | 6x6 | 6x1 | 0.02 |
| integrate | Integrate | Map/Gather | 2D/3D | 3D | 12.85 |
| raycast | Raycast | Search/Stencil | 2D/3D | 2D | 35.87 |
| renderDepth | Rendering | Map | 2D | 2D | 0.12 |
| renderTrack | Rendering | Map | 2D | 2D | 0.06 |
| renderVolume | Rendering | Search/Stencil | 3D | 2D | 9.18 |

TABLE I: SLAMBench kernels. *Pipeline* refers to SLAM high-level building blocks of the top row of Fig 2. *Pattern* describes the parallel computation pattern (see sec. IV-C) of a kernel. % gives the percentage of the time spent in the kernel while running sequentially on the Exynos 5250 processor (see platforms Table II). For the breakdown in other platforms see Fig. 5.

the calculation of a running average over the current TSDF volume and the new depth image.

KinectFusion has been adopted as a major building block in more recent SLAM systems [27], [15], [28], [8], [20] and is part of Microsoft's Kinect SDK [2]. A number of open implementations of the KinectFusion algorithm have been made in recent years, including KFusion [21] (utilised in this paper) and KinFu [22].

## IV. SLAMBENCH

SLAMBench, see Fig. 2, is a benchmark that provides portable, but untuned, KinectFusion [19] implementations in C++ (sequential), OpenMP, CUDA and OpenCL for a range of target platforms. As an example, the ODROID board contains two GPU devices, see Sec. V-A, but the current OpenCL implementation only exploits one GPU device. SLAMBench includes techniques and tools to validate an implementation's accuracy using the ICL-NUIM dataset along with algorithmic modifications to explore accuracy and performance trade-offs.

### A. ICL-NUIM DATASET

ICL-NUIM [12] is a high-quality synthetic dataset providing RGB-D sequences for 4 different camera trajectories through a living room model. An *absolute trajectory error* (ATE) is calculated as the difference between the ground truth and the estimated trajectory of a camera produced by a SLAM implementation, this enables system accuracy to be measured at each frame.

Fig. 1 shows the input scene from the ICL-NUIM dataset and the reconstruction performed using SLAMBench. ICL-NUIM can provide frames sequences that: (1), are free of

noise, and (2), have noise added to RGB-D frames according to a statistical model matching Microsoft's Kinect sensor. The noise model is applied to each frame used as input to a SLAM system in order to give realistic results.

The ICL-NUIM benchmark provides not only a number of realistic pre-rendered sequences, but also open source code that can be used by researchers in order to generate their own test data as required. As the unmodified ICL-NUIM dataset is the input of the SLAMBench benchmark any user can run SLAMBench in a straightforward way on other datasets.

### B. SLAMBENCH KERNELS

Table I summarises the 14 main computationally significant computer vision kernels contained in SLAMBench. The following is a description of each kernel:

1) **acquire**: acquires a new RGB-D frame. This input step is included explicitly in order to account for I/O costs during benchmarking, and for real applications.
2) **mm2meters**: transforms a 2D depth image from millimeters to meters. If the input image size is not the standard 640x480, only a part of the image is converted and mapped into the output.
3) **bilateralFilter**: is an edge-preserving blurring filter applied to the depth image. It reduces the effects of noise and invalid depth values.
4) **halfSample**: is used to create a three-level image pyramid by sub-sampling the filtered depth image. Tracking solutions from low resolution images in the pyramid are used as guesses to higher resolutions.
5) **depth2vertex**: transforms each pixel of a new depth image into a 3D point (vertex). As a result, this kernel generates a point cloud.
6) **vertex2normal**: computes the normal vectors for each vertex of a point cloud. Normals are used in the projective data association step of the ICP algorithm

to calculate the point-plane distances between two corresponding vertices of the synthetic point cloud and a new point cloud.

7) **track**: establishes correspondence between vertices in the synthetic and new point cloud.

8) **reduce**: adds up all the distances (errors) of corresponding vertices of two point clouds for the minimisation process. On GPUs, the final sum is obtained using a parallel tree-based reduction.

9) **solve**: uses TooN [5] to perform a singular value decomposition on the CPU that solves a 6x6 linear system. A 6-dimensional vector is produced to correct the new estimate of camera pose.

10) **integrate**: integrates the new point cloud into the 3D volume. It computes the running average used in the fusion described in Sec. III.

11) **raycast**: computes the point cloud and normals corresponding to the current estimate of the camera position.

12) **renderDepth**: visualises the depth map acquired from the sensor using a colour coding.

13) **renderTrack**: visualises the result of the tracking. For each pixel different colours are associated with one of the possible outcomes of the tracking pass, e.g. 'correct tracking', 'pixel too far away', 'wrong normal', etc.

14) **renderVolume**: visualises the 3D reconstruction from a fixed viewpoint (or a user specified viewpoint when in the GUI mode).

All kernels are implemented on GPUs, in OpenCL and CUDA, except *acquire* and *solve*.

In addition to these kernels, the SLAMBench pipeline also contains two initialisation kernels not shown in Table I and Fig. 2: *generateGaussian* and *InitVolume*. These kernels are part of an initialisation step that is performed only at start-up. *generateGaussian* generates a Gaussian bell curve and stores it in a 1D array; *initVolume* initialises the 3D volume.

### C. SLAMBENCH PARALLEL PATTERNS

The parallel patterns of Table I are defined following definitions from [18] and shown in Fig. 3. Since the kernels represent complex real-world SLAM computations, the computation of a single kernel may fit multiple parallel patterns as depicted in Table I.

- **Map**: (see Fig. 3(a)), a pure function operates on every element of an input array and produces one result per element. Threads do not need to synchronise.
- **Reduction**: (see Fig. 3(b)), a reduction function combines all the elements of an input array to generate a single output. In this case, tree-based implementations can be used to parallelise such a reduction.
- **Stencil**: (see Fig. 3(c)), each output element is computed by applying a function on its corresponding input array element and its neighbors.
- **Gather**: (see Fig. 3(d)), is similar to map but its memory accesses are random. Parallel gather implementations are similar to those of map.
- **Search**: (see Fig. 3(e)), is similar to gather, but it retrieves data from an array based on matching against

content. Parallelism is achieved by searching for all keys in parallel, or by searching in different parts of the array in parallel.

### D. PERFORMANCE EVALUATION METHODOLOGY

A methodology for SLAM performance evaluation is necessary for three reasons. First, since we are dealing with numerical approximations and iterative algorithms a systematic way to check the accuracy of the results on different platforms is required. For example, in the pose estimation, KFusion assumes small motion, through a small angle approximation, from one frame to the next in order to achieve fast projective data association [6]. Also, the rigid body transform in the camera pose estimation is computed by testing whether each new iteration carries new useful information. Finally, the error between point clouds is computed using a reduction of $l^2 - norms$, whose result may depend on the order in which summation takes place.

The second issue requiring careful evaluation methodology is that we are dealing with an application that may be used in different contexts, such as mobile robotics and AR. Each application places different stress on the SLAM pipeline.

Thirdly, the KinectFusion computation is dependent on the images acquired, the way the camera is moved and the tracking frame rate. A methodology is required to enable a SLAMBench user to avoid divergent behaviors across different platforms where the target system cannot process all frames in real-time. We use a dataset of pre-recorded scenes and the *Process-every-frame* mode described below.

The SLAMBench methodology consists of four elements:

*1) GUI vs terminal interface:* SLAMBench includes raycasting for visualisation but we have not included the display of the image. Rendering using OpenGL lies outside the scope of this work.

*2) Pre-recorded scenes:* The KinectFusion computation depends upon the scene and camera trajectory. In order to address the variability that comes with the images and the camera movement in the benchmark space we use pre-recorded scenes described in Sec. IV-A, thus ensuring reproducibility.

It is important to notice that using pre-recorded scenes can impact the performance of the application since in this case the *acquire* is negligible compared to other kernels, as shown in Sec. V-B. But in a real-life context the input device induces unavoidable latency, which makes it important to take this into account in the performance analysis.

*3) Frame rate:* in a real-time application context, the frame rate of the application depends on the computational speed of the platform used. If the application is unable to process frames in real-time then input frames are dropped. To ensure consistent evaluation of accuracy, energy consumption and performance it is necessary to guarantee that the compared applications process the same set of frames. For this reason we introduce a *Process-every-frame* mode, where the application acquires the next pre-recorded frame (without considering timestamps) from a dataset file only when the previous frame has been processed. The frames are
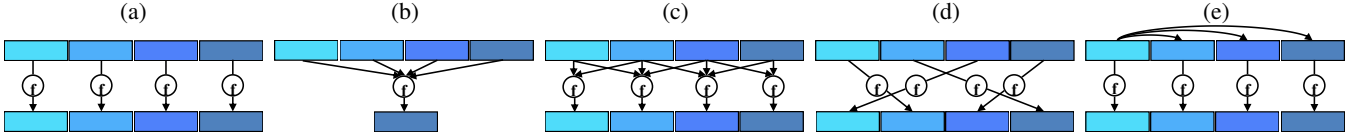
Fig. 3: Parallel patterns: (a) Map, (b) Reduction, (c) Stencil, (d) Gather and (e) Search.

enumerated and scheduled in same order on all platforms. This ensures that all simulations process exactly the same frames thus rendering the computation independent of the frequency of the camera and performance of the platform.

*4) Accuracy evaluation:* SLAMBench provides an automatic tool that performs testing against the ground truth from ICL-NUIM. The ATE, mean, median, standard deviation, min and max statistics errors (as described in IV-A) are computed using the synthetic trajectories given by the ICL-NUIM. This allows us to ensure that versions of the implementation using different languages and platforms are sufficiently accurate, whilst not expecting bit-wise output equivalence.

We note SLAMBench currently only exploits the trajectory ground truth from ICL-NUIM, and not the 3D reconstruction it also offers. This is because, as described by Handa et al. [12], interactively driven computer vision tools are required in order to perform reconstruction comparison. However, localisation and mapping are strongly correlated, which means that the ATE serves as a good indication of whether the reconstruction is accurate enough.

## V. EXPERIMENTAL EVALUATION

### A. DEVICES

For our experiments we used five platforms, Table II summarises their characteristics. For the sake of brevity we refer to these machines as TITAN, GTX870M, TK1, ODROID and Arndale. SLAMBench is targeted at processors from the desktop down to low power arenas, e.g. processors used in smart-phones and tablets. These embedded devices are gaining in popularity and acquiring more capable cameras and mobile processors. We carefully select the platforms to be representative of the actual devices used by consumers and industry in the different desktop, mobile and embedded categories.

Comparison between different GPU architectures is a non trivial task; this is not helped by the wide range of vendor specific languages used, commercial sensitivity and varying architectures. NVIDIA characterise their devices by the number of "CUDA cores". Thus the TK1 has 192 CUDA cores while ARM's GPUs have 4-8 more powerful cores.

In the ARM embedded devices space we run experiments on the Hardkernel ODROID-XU3 based on the Samsung Exynos 5422. The Exynos 5422 implements ARM's big.LITTLE heterogeneous multiprocessing (HMP) solution with four Cortex-A15, "big" out of order processors, four "LITTLE" in order processors and a Mali-T628-MP6 GPU; this GPU essentially consists of two separate devices one consisting of 4 cores and the other 2. In our experiments we only use the 4-core GPU device. The ODROID platform has integrated power monitors which will be explored in Sec V-C. At the ARM end we also perform experiments on the dual-core Arndale board based on a Samsung Exynos 5250 and MALI-T604-MP4 GPU.

Jetson TK1 is NVIDIA's embedded Linux development platform featuring a Tegra K1 SoC. It is provided with a quad-core ARM Cortex-A15 (NVIDIA 4-Plus-1) in the same SoC with an NVIDA GPU composed of 192 CUDA cores.

At the desktop end of the spectrum we selected a 4-core Intel Haswell i7-4770K associated with an NVIDIA GTX TITAN GPU containing 2688 CUDA cores. This machine is a high-end desktop with a power envelope up to 400 W, although the actual power consumption is, in our experience, considerably less. Including such a powerful machine in the testbed is interesting in order to explore the limits of the application when many parallel compute units are available.

In order to be as representative as possible on the cross platform evaluation, we also include a high-end laptop. The laptop features an Intel Haswell mobile processor i7-4700MQ together with an NVIDIA GTX 870M mobile discrete GPU. This GPU contains 1344 CUDA cores, making it more suitable than the GTX TITAN for high-end laptop usage.

### B. PERFORMANCE AND ACCURACY RESULTS

Default SLAMBench parameters are used for the experiments as in [21]: integration rate is set at 2 frames, the voxel volume is 256x256x256, the volume size is $4.8x4.8x4.8 \ m^3$, rendering rate is set at 4 frames and the input image is halved during the preprocessing, i.e. 320x240. All the experiments are run on the living room trajectory 2 of the ICL-NUIM dataset. Table III demonstrates a consistent maximum ATE of between 2.01cm on the ODROID OpenCL implementation, and 2.07 cm on other platforms/implementations was computed over all runs. These errors are comparable with [12]. The *reduce* kernel gives rise to small numerical differences as it performs non-associative summations leading to different rounding errors. Fig. 4 shows the performance achieved across different platforms for all available languages. The FPS achieved are reported on top of each histogram. The y axis represents the time in seconds to process one frame. The TITAN desktop achieves the highest frame rate running at 135 FPS. Using their on-die GPU, the embedded TK1 platform almost reaches real-time with ca. 22 FPS, while the ODROID and Arndale boards achieve an average frame rate of 5.46 and 4.21 respectively. Using the CPU cores only, via OpenMP, the 8-core ODROID platform achieves 2.76 FPS while the Arndale and TK1 reach 0.8 and 1.78 respectively.

| Machine names | TITAN | GTX870M | TK1 | ODROID (XU3) | Arndale |
|---|---|---|---|---|---|
| Machine type | Desktop | Laptop | Embedded | Embedded | Embedded |
| CPU | i7 Haswell | i7 Haswell | NVIDIA 4-Plus-1 | Exynos 5422 | Exynos 5250 |
| CPU cores | 4 | 4 | 4 + 1 (Cortex-A15) | 4 + 4 (Cortex-A7 and A15) | 2 (Cortex-A15) |
| CPU GHz | 3.5 | 2.4 | 2.3 | 1.8 | 1.7 |
| GPU | NVIDIA TITAN | NVIDIA GTX 870M | NVIDIA Tegra K1 | ARM Mali-T628-MP6 | ARM Mali-T604-MP4 |
| GPU architecture | Kepler | Kepler | Kepler | Midgard 2nd gen. | Midgard 1st gen. |
| GPU FPU32s | 2688 | 1344 | 192 | 60 | 40 |
| GPU MHz | 837 | 941 | 852 | 600 | 533 |
| GPU GFLOPS (SP) | 4500 | 2520 | 330 | 60+30 (72+36) | 60 (71) |
| Language | CUDA/OpenCL/C++ | CUDA/OpenCL/C++ | CUDA/C++ | OpenCL/C++ | OpenCL/C++ |
| OpenCL version | 1.1 | 1.1 | n/a | 1.1 | 1.1 |
| Toolkit version | CUDA 5.5 | CUDA 5.5 | CUDA 6.0 | Mali SDK1.1. | Mali SDK1.1 |
| Ubuntu OS (kernel) | 13.04 (3.8.0) | 14.04 (3.13.0) | 14.04 (3.10.24) | 14.04 (3.10.53) | 12.04 (3.11.0) |

TABLE II: Devices used for our experiments.

| TITAN | | | | GTX870M | | | | TK1 | | | ODROID | | | Arndale | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C++ | OMP | OCL | CUDA | C++ | OMP | OCL | CUDA | C++ | OMP | CUDA | C++ | OMP | OCL | C++ | OMP | OCL |
| 2.07 | 2.07 | 2.07 | 2.07 | 2.07 | 2.07 | 2.07 | 2.07 | 2.06 | 2.06 | 2.07 | 2.06 | 2.06 | 2.01 | 2.06 | 2.06 | 2.07 |

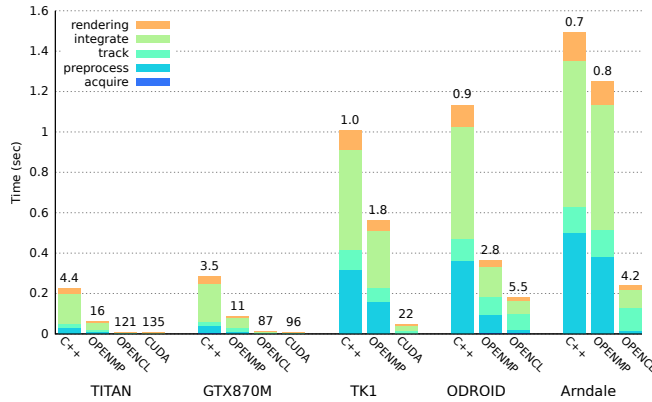TABLE III: Absolute trajectory error (ATE) in centimeters.



Fig. 4: High-level SLAM building block elapsed times. FPS is reported on top of each histogram.



Fig. 5: Percentage of time spent in each kernel.

Despite our efforts to make the OpenCL and CUDA versions as similar as possible, we observe a 10% difference of performance in favour of CUDA. An unexpected result is the difference in the proportion of time spent in each kernel between OpenCL and CUDA in the desktop and laptop histograms, see next paragraph. We are currently investigating why such a wide difference exists and why *renderDepth* and *renderTrack* scale so badly in CUDA.

Fig. 5 shows the percentage of time spent in each kernel and how this changes over different platforms/implementations. Ideally, an application containing kernels that scale perfectly will have the same proportions on all the histograms of a given platform. If a kernel scales well, i.e. better than other kernels in the average, then it would reduce its proportion on a histogram that contains more computational units for a given platform. As an example, if we consider the 8-core ODROID and we compare the C++ and OpenMP histograms we can see that the *bilateralFilter* and the *raycast* scales well. In contrast the *reduce* and the
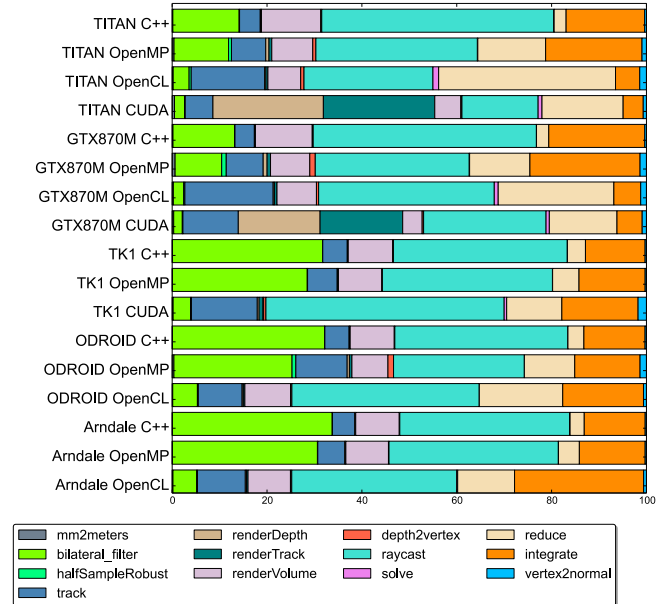
*track* kernels scale badly, whereas the *integrate* is average. We can see this effect on all the platforms. The proportions stay roughly unchanged on the 2-core Arndale because of the limited number of cores.

There is a correlation between Table I and the qualitative scaling factors observed in Fig. 5. The reasons why *bilateralFilter* scales well are threefold: first it operates a stencil pattern; second it loads/stores 2D images which implies that data can fit into caches exploiting data locality; third the percentage of the program taken by this kernel is not negligible which means that this kernel has room for speedup. The *reduce* kernel does not scale linearly because it follows a reduce pattern; despite tree-based reduction, this kind of pattern suffers on highly parallel devices as can

be observed in the CUDA and OpenCL histograms. Since *integrate* manipulates 3D data it takes a performance hit on platforms with reduced data cache size, i.e. embedded devices, while scaling nicely on TITAN and GTX780M. The *raycast* bar is approximately constant across the histograms. This kernel also manipulates 3D data and, as a consequence, has high memory bandwidth demand and has intensive computation due to the stencil pattern it operates. The high number of load/store operations is balanced by the high computation needed.

We can also notice that small kernels, which are not visible in the C++ histograms, if they scale badly, will appear in histograms where massive parallel resources are deployed. An example is the CPU-only *solve* kernel.

### C. ENERGY CONSUMPTION ANALYSIS: ODROID-XU3 CASE STUDY

Anecdotally it is sometimes suggested that GPU based applications are power-hungry, indeed in many systems they do draw a significant amount of the system power, however the GPU may also carry a disproportionate amount of the workload. We use the ODROID device introduced in Sec. V-A to evaluate the energy efficiency of its various processing elements in the context of a SLAM application. The platform has on-board voltage/current sensors and split power rails, which allows the power consumption of the "big" Cortex-A15 cores, the "LITTLE" Cortex-A7 cores, GPU and DRAM to be measured individually. The sensors can be read by the SLAMBench application to provide measurements at various points of execution which can be logged or plotted interactively in the GUI. Energy consumption is only explored on the ODROID because consistent power analysis tools are not present on all devices.

We use SLAMBench along with the platform's energy monitors to measure energy/power consumption and performance using the C++, OpenMP and OpenCL implementations; for C++ tests using both Cortex-A15 and Cortex-A7, for OpenMP using all eight cores or just Cortex-A7 and OpenCL using only the larger GPU device and either a Cortex-A15 or Cortex-A7. These tests are conducted using Process-every-frame mode, and consequently show the relative energy efficiency of the various implementation/resource combinations for the same amount of computation. Currently, the ODROID's Linux kernel only supports a performance driver which prefers to shift threads onto the Cortex-A15 cores, to deliver high performance. In order to prevent this, and to force the use of Cortex-A7 cores, we set the Cortex-A15 cores to be offline where appropriate.

Fig. 6 clearly demonstrates that the OpenCL/GPU implementation uses significantly less energy than approaches that rely solely on the conventional CPU cores. Furthermore we can see that, for OpenCL, the performance benefit from using the Cortex-A15 over the Cortex-A7 is negligible, but the energy consumed is 50% higher (507.28 compared with 337 Joules), there is consequently no benefit in this application of using the high performance CPU. Implementation using OpenMP on Cortex-A7 also gives an improvement in energy
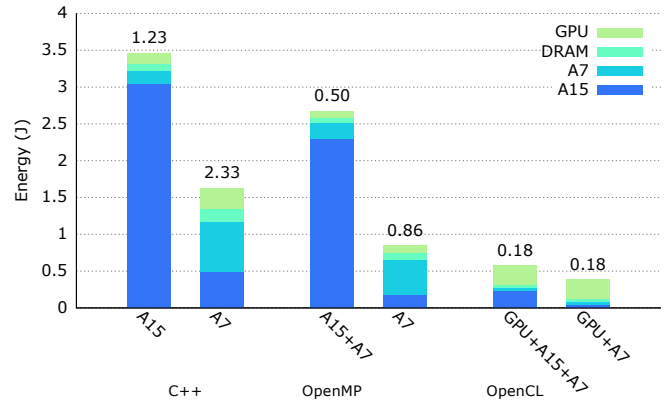


Fig. 6: Energy usage per frame utilising various hardware resources on the ODROID, mean time per frame marked

over a single core C++ implementation, assuming a race to sleep strategy is employed whereby data is processed as fast we can then we switch a low power mode, indicating potential energy savings are achievable in SLAM applications by using parallel implementations. There is a significant energy usage for the Cortex-A15 even when it has been configured as offline, this would point to the operating system not forcing the cores to be powered down and indicating a greater reduction in energy consumption may be achievable when they are not in use.

A SLAM system will fail if the inter-frame movement of the camera is too great, as it will fail to track the camera pose. We can see for the ODROID we would typically drop a significant number of frames using our default settings, of voxel size, image size, integration rate and rendering rate. SLAMBench allows the user to run at the rate of the system, dropping frames where appropriate. The user can then explore the design space to achieve an optimal ATE. For example on an embedded system they may choose not to create a coarser model, render our model as it is created - this is typical for a robotics application - or update the model less often, all of which will, typically, deliver substantial reduction in dropped frames and, within limits, the ATE.

Parallel approaches offer a significant energy, and performance, improvement over single threaded approaches. The Mali GPU provides a significantly improved energy efficiency when compared with conventional cores for this SLAM application, but care must be taken to ensure that conventional CPU threads are scheduled on the most appropriate conventional cores, and that unused cores are put into an appropriate low power state.

## VI. CONCLUSIONS AND FUTURE WORK

SLAMBench is the first SLAM benchmark which is concerned not just with accuracy but also computational performance and energy consumption. We show how SLAMBench allows a user to perform a fair comparison of hardware accelerators, software tools and novel algorithms in a modern dense SLAM framework, where our framework for automatically comparing estimated trajectories against ground truth

ensures that estimation accuracy is maintained and separated from performance comparison. Each SLAMBench kernel is characterised by its parallel pattern and the weight it has in the SLAM computation; swapping module implementations is relatively straightforward so that certain parts of an algorithm can be isolated and improved.

We present performance results on state-of-the-art desktop, laptop and embedded devices. Four configurations achieved super real-time FPS in the desktop and laptop space with a peak performance of 135 FPS. The Tegra K1 achieves 22 FPS, which is impressive for an embedded device, though this board is not optimised for energy usage, which may restrict its applicability in robotics. In contrast, the ODROID-XU3 achieves 5.5 FPS for 2.1 Watts average dissipation. All of these performance figures leverage the efficiency of GPUs, showing how effective heterogeneous computing can be for SLAM on a wide range of modern devices.

We have also presented the first SLAM energy consumption evaluation on the ODROID-XU3 embedded platform which is suitable for robotics, analysing the performance/power trade-off at a fixed accuracy. The Cortex-A7/MALI provides the same FPS as the Cortex-A15/MALI with a 50% lower energy envelope. This research paves the way for systematic power dissipation evaluation using SLAMBench. Future work will include the addition of performance counter based energy monitoring, on devices that provide hardware counter APIs, by inclusion of PAPI within the framework as outlined in [26]

SLAMBench facilitates design-space exploration. A number of algorithm parameters, such as the voxel density or the TSDF distance $\mu$, are available in the SLAMBench user interface. Exploring how these parameters improve the SLAM application while simultaneously changing compiler optimisation parameters, such as the OpenCL work-group size or thread-coarsening [17], may have a high impact on performance/power/accuracy metrics.

Finally, SLAMBench provides the necessary foundation for investigating domain-specific optimisations. We plan to build our own DSL, targeting high performance, low-power SLAM applications, and believe that it will inspire similar efforts from relevant communities. SLAMBench is intended to develop over time and it is expected that it will include a larger set of SLAM kernels in later versions. A key step would be to integrate kernels that allow KFusion to be more scalable in terms of the size of the scene to be reconstructed, for example, by adding alternative and complementary data structures such as point-based fusion [15], octrees [28], voxel hashing-based representation [20] and moving volumes, as used in Kintinous [27].

## ACKNOWLEDGMENTS

## REFERENCES

[1] Dyson 360 Eye web site. https://www.dyson360eye.com.
[2] Microsoft Kinect SDK programming guide. http://msdn.microsoft.com/en-us/library/dn188670.aspx.
[3] Project Tango web site. https://www.google.com/atap/projecttango.
[4] SLAMBench web site. http://apt.cs.manchester.ac.uk/projects/PAMELA/tools/SLAMBench.
[5] TooN web site. http://www.edwardrosten.com/cvd/toon/html-user/.
[6] P. J. Besl and N. D. McKay. Method for registration of 3-D shapes. In *Robotics-DL tentative*. Int. Society for Optics and Photonics, 1992.
[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, 2006.
[8] J. Chen, D. Bautembach, and S. Izadi. Scalable real-time volumetric surface reconstruction. *ACM TOG*, 2013.
[9] J. Clemons, H. Zhu, S. Savarese, and T. Austin. MEVBench: A mobile computer vision benchmarking suite. In *IISWC*, 2011.
[10] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proc. Computer graphics and interactive techniques*. ACM, 1996.
[11] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *CVPR*, 2012.
[12] A. Handa, T. Whelan, J. McDonald, and A. Davison. A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM. In *ICRA*, 2014.
[13] S. J. Henderson and S. K. Feiner. Augmented reality in the psychomotor phase of a procedural task. In *ISMAR*, 2011.
[14] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.
[15] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb. Real-time 3D reconstruction in dynamic scenes using point-based fusion. In *3DTV-Conference*. IEEE, 2013.
[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. ACM/IEEE Int. Symp. on Microarchitecture*, 1997.
[17] A. Magni, C. Dubach, and M. F. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proc. of SC13: Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.
[18] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proc. of the 2nd USENIX conf. on Hot topics in parallelism*, 2010.
[19] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *ISMAR*, 2011.
[20] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM TOG*, 2013.
[21] G. Reitmayr and H. Seichter. KFusion github. https://github.com/GerhardR/kfusion, 2011.
[22] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *ICRA*, 2011.
[23] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of RGB-D SLAM systems. In *IROS*, 2012.
[24] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth Int. Conf. on Computer Vision*. IEEE, 1998.
[25] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *IISWC*, 2009.
[26] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring Energy and Power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268, Sept 2012.
[27] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald. Kintinuous: Spatially Extended KinectFusion. In *WS on RGB-D: Advanced Reasoning with Depth Cameras*, 2012.
[28] M. Zeng, F. Zhao, J. Zheng, and X. Liu. Octree-based fusion for realtime 3D reconstruction. *Graphical Models*, 2013.