Performance Evaluation and Optimisation for Dense 3D Scene Understanding using SLAMBench

Yi Kong[1]
yk1211@imperial.ac.uk

Imperial College London

16th June 2015

[1]Supervisor: Dr. Luigi Nardi, Prof. Paul Kelly; Second marker: Dr. Tony Field

Abstract

One of the biggest drivers for future multi-core computing will be computer vision. A key issue is optimisation for power, as well as performance. In particular, equipping robots, sensors and wearable devices with 3D scene understanding will become a platform for a huge range of applications. To help study this, a dense 3D vision benchmark called SLAMBench was developed that enables studying algorithmic and implementation choices and measure the impact not only on performance/power, but also end-to-end quality of results. SLAMBench is based on an implementation of the KinectFusion algorithm.

The goal of this project is to explore different implementations of the KinectFusion kernels and investigate where and how to introduce optimisations of various kinds (SIMD vectorisation, memory alignment, structure of arrays vs array of structs, loop transformations, etc.) for a modern embedded multi-processor system, explore the hardware design space to further increase performance while reducing energy drain as well as die area. We achieved 21% improvement in throughput through guided optimisation and 40% in energy-area product through alternative processor designs.

Contents

List of Tables

List of Figures

List of Source Codes

Chapter 1

Introduction

Computer Vision is gaining popularity recently in all fields of science and engineering and simultaneous localisation and mapping (SLAM) is one of the most important domains because it is the central challenge in facilitating navigation in previously unexplored environments in robotics and augmented reality (AR).

SLAM is the problem of building a map of the environment while simultaneously localising the agent within that map. At first it may seems to be a chicken-and-egg problem where a map is needed to localize while a pose estimate is required for mapping, however there are several algorithms developed to solve it with probabilistic concepts. It is regarded as one of the most challenging problem in computer vision, a 2008 review of the topic [1] summarised: "[SLAM] is one of the fundamental challenges of robotics . . . [but it] seems that almost all the current approaches can not perform consistent maps for large areas, mainly due to the increase of the computational cost and due to the uncertainties that become prohibitive when the scenario becomes larger."

In the last decade the progress made in computer vision has been astounding. Point feature-based SLAM techniques are developed to drastically reduced the computational requirement, and they are now present in many commercial embedded systems, including Dyson 360 [2], Google's Project Tango [3] and its SCHAFT man-bot. However dense SLAM is still in early prototype state. Several real-time algorithms for dense 3D scene reconstruction are developed in the past few years such as PTAM [4], KinectFusion [5] and StereoScan [6], which estimates the pose of the camera while building a highly detailed 3D model of the static environment using a specialized camera. Such capabilities can allow robots to interact with the world in ways we have never imagined before.

1.1 Motivations

There has been a great focus to develop a specialised benchmarking system to aid the research in computer vision.

SLAMBench [7] is the first performance, energy and accuracy benchmark dedicated to 3D scene understanding applications. It allows researchers to explore the design space of algorithmic and implementation-level optimisations in dense SLAM. It contains a portable, but untuned, KinectFusion [5] implementation in C++ with OpenMP, OpenCL and CUDA for a wide range of target systems. It provides tools to measure the time and energy usage of the implementation and the accuracy comparing with ground truth using the ICL-NUIM [8] dataset.

KinectFusion [5] is the state-of-the-art algorithm for dense 3D scene reconstruction and is now widely used in commercial products like Microsoft XBox and many Windows Games. As the adoption is now extending to embedded systems like HoloLens [9], it became increasingly important that we understand its behaviour on various target systems and able to achieve real-time power-efficient on systems with restricted computing resources.

1.2 Objectives

The aim of this project is to explore KinectFusion kernels within SLAMBench and investigate the characteristics of a selection of the most important kernel components, and use various analyses and the Roofline model [10] to guide optimisation of the kernels and evaluate the results on a modern embedded system, using actual hardware combined with gem5 [11] simulator system. Finally we explore the hardware design-space to enable more efficient computation, in order to achieve real-time reconstruction solely using CPU.

In Chapter 4 we provide detailed dynamic runtime analysis of KinectFusion kernels, providing their dynamic instruction distribution, cache behaviour and branch prediction using real hardware as well as the gem5 simulator system.

In Chapter 5, we attempt to implement the optimisations from opportunities we discovered and benchmark the achieved improvement and compare with the theoretical limit. We then evaluate the difficulty and effectiveness of our methodology.

In Chapter 6, with the insights gained from previous parts, we look into exploring the design space, tweaking hardware implementation to accelerate the SLAMBench, to enable fast, energy efficient, economic computation platform, achieving real-time energy-efficient detailed 3D reconstruction relying solely on a multi-core CPU.

1.3 Contributions

The main contributions of this thesis are:

- We present a detailed dynamic analysis of a KinectFusion [5] algorithm implementation on ARM Cortex-A15 architecture.
- We demonstrate a systematic approach to use various dynamic analysis and the Roofline [10] model to perform guided optimisation of SLAMBench kernels with the combination of real hardware and gem5 [11] simulator. We show that we achieved 21% improvement in throughput on Samsung Chromebook "snow" and we evaluate the effectiveness of the technique in terms of human effort.
- We devise a process to utilise dynamic analysis to greatly reduce hardware design exploration space and show that we can reduce delay-area product by 40% through a GPGPU-like CPU design.

Our work gives a systematic way to optimise arbitrary computer vision kernels with low human effort and allows researchers to explore the design space of algorithms in dense SLAM more effectively.

Chapter 2

Background

Our work is established on the base of many computer architectural research tools and models. In this chapter, we briefly introduce them and their contribution to our work. Section 2.1 introduces KinectFusion, the algorithm we are optimising in the thesis. Section 2.2, we introduce the Roofline model, a performance model for floating-point programs and multi-core CPU architectures. In section 2.4, we talk about the tradeoffs we make when designing a processor architecture, including energy, performance and die area, leading to 2.5 where we introduce McPAT, the model we use to analyse the processor design.

2.1 KinectFusion

KinectFusion [5] is a system for dense volumetric reconstruction of complex indoor scenes, using only a low-cost, noisy monocular depth camera, such as Microsoft Kinect [12]. It can track 6 degress-of-freedom (6DOF) pose of the sensor while generates a continuously updating, smooth, fully-fused 3D surface reconstruction. It fuses the depth data from the specialized sensor into a single global 3D reconstruction model of the observed scene in real-time. The current sensor pose is simultaneously obtained by tracking the subsequent depth image frame relative to the current global reconstruction model using a coarse-to-fine multi-scale iterative closest point (ICP) algorithm using all the available data at frame-rate. The algorithm is optimised for highly parallel general purpose GPU (GPGPU), allowing both tracking and mapping in real time while consuming small amount of energy. Figure 2.1 shows an example output from KinectFusion generated with a hand-held Kinect camera.

Although the details of the algorithm and the mathematical concepts are beyond of scope for our work, we briefly outline the key computational steps involved. The KinectFusion algorithm is comprised of the following four components:

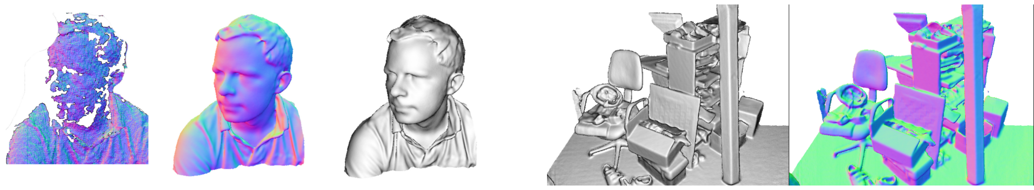


Figure 2.1: Example output from KinectFusion. [5]

Surface Measurement

A pre-processing stage. A bilateral filter [13] is applied to the raw depth map obtained from the sensor to reduce its noise. An $L = 3$ level multi-scale representation of the surface is then computed, in a form of dense vertex map and normal map pyramid.

Mapping as Surface Reconstruction

The global scene fusion process, where each consecutive raw depth frame is fused incrementally with the associated sensor pose estimate into a 3D reconstruction using a variant of iterative closest point (ICP) algorithm [14].

Surface Prediction from Ray Casting the TSDF

The loop between mapping and localisation by tracking the live depth frame against the globally fused 3D reconstruction is closed by raycasting the volumetric, truncated signed distance function (TSDF) [15] into the estimated frame to provide a dense 3D surface prediction, where the surfaces are estimated at the zero crossings of the function.

Sensor Pose Estimation

Live sensor localisation using a multi-scale ICP alignment between the model prediction from previous frame and live surface measurement, by assuming small inter-frame sensor motion.

All components of KinectFusion, including tracking and mapping, have trivially parallelisable structure and scales efficiently with processing power, memory bandwidth and memory size, taking advantage of the parallelism of commodity multi-core or GPGPU processing hardware.

As an optimisation to performance, the algorithm may skip surface update given that the tracking is accurate enough. Also on failure to track the sensor, the algorithm will perform re-localisation, where the last known sensor pose is used to provide a surface prediction. They both lead to data dependent running time, which we will further discuss in the later chapters.

Figure 2.2 shows the complete work-flow of KinectFusion components.

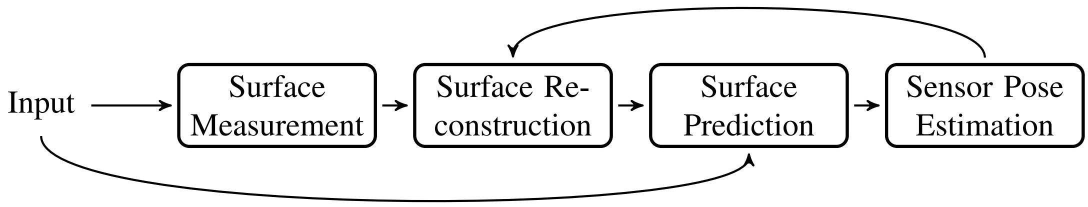


Figure 2.2: KinectFusion components workflow.

2.2 Roofline Model

Roofline [10] is a performance model for floating-point programs and multi-core CPU architectures, aiding programmers and hardware architects to improve parallel software and hardware for floating point computations. It uses *Bound and Bottleneck analysis* to provide insights into the factors that are negatively affecting the performance of program. It mainly focuses resources including memory bandwidth and raw computation power.

To model the memory bandwidth an application uses, Roofline uses *operational intensity*, meaning (floating) operations per byte of data traffic. The data traffic is limited to the data transferred between cache hierarchy and off-chip DRAM memory, since cache accesses are faster in orders of magnitude and it is often the constraining resource [16]. Thus *operational intensity* suggests the minimum DRAM bandwidth required for a particular system.

Roofline also models the peak floating-point performance in terms of the number of floating-point operations the system can execute per second (FLOPS). This can be found from hardware specification, but also obtainable by running micro-benchmarks like the STREAM benchmark [17].

Figure 2.3 shows an example Roofline Model for a AMD Opteron X2, with data obtained from running STREAM micro-benchmark.

The model gives an upper bound to performance achievable by an arbitrary kernel. To further utilize the model, "performance ceilings" where implementations without different optimisations will be limited by the corresponding limit. For example, the following optimisations can be performed on kernels to reduce bottlenecks.

Computational Bottlenecks:

ILP or SIMD

Instruction Level Parallelism (ILP) works by arranging instructions so that there is no data dependency in adjacent instructions, so that the super-scalar architectures can fetch, execute and commit multiple instructions simultan-

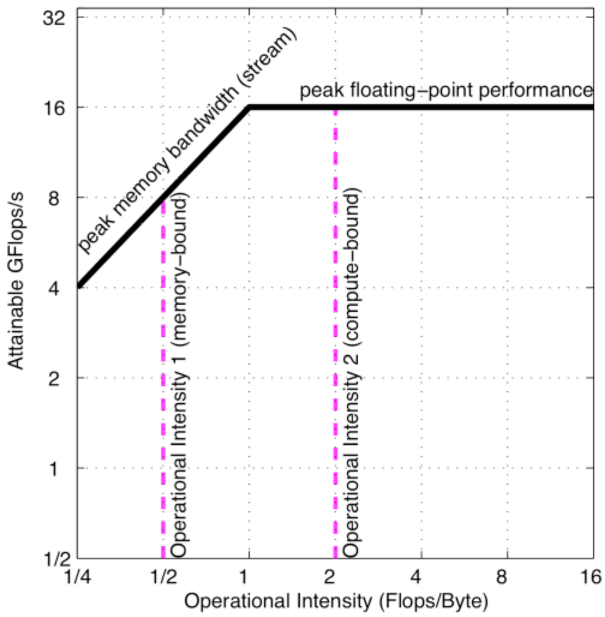


Figure 2.3: Roofline Model for Opteron X2. (diagram is from [10])

eously.

Single instruction, multiple data (SIMD), is a special class of commands where the CPU executes same instructions on multiple data. These operations can usually be completed in a single cycle, resulting in performance increase of up to several folds.

Floating-point Operation Balance

Most modern CPU ISAs supports multiply–accumulate operation, which is executed by the multiplier-accumulator (MAC) unit. However, in order to take advantage of such instructions, floating-point add and multiply instructions must be balanced and interleaved. For machines that do not possess such unit, they usually can still be benefited from such optimisations because the number of floating-point adder and multiplier units are usually similar.

Memory Bandwidth Bottlenecks:

RestrctureLoops for Unit Stride Accesses

Hardware cache and data pre-fetching are optimised for read or write to continuous regions of memory. Rewriting loops to facilitate such optimisations can reduce the data being transferred between cache hierarchies and memory system and significantly increases memory bandwidth.

Memory Affinity

This is an optimisation limited to Non-uniform memory access (NUMA) systems, where accessing local regions of memory is substantially faster. By making threads access remote memory less frequently, the memory bandwidth could be improved.

Software Prefetching

When accessing a continuous region of data, the CPU is able to predict and prefetch data quite accurately. However for random access, the use of prefetch instruction can teach the processor to load data into cache in advance thus reducing the memory access latency and improving total memory throughput.

Figure 2.4 adds ceilings to the Roofline model for Opteron X2, employing the optimisations mentioned above.

Existing un-tuned kernels can be plotted onto the model by running the kernel and registering the performance counters. The position of the kernel in the model suggests the percentage of hardware capability it is currently using. If the kernel sits directly on the computational resources "roofline" means it is executing with full hardware capabilities. If plotted on the model with optimisation ceilings, the potential optimisation opportunities can be easily identified and programmers can attempt to utilise the optimisation directly on the code. Figure 2.5 shows the general ways to optimise in Roofline model.

2.3 gem5 Simulator System

The gem5 simulator [11] is a state of the art tool for computer system architecture research, widely used in academia and industry, encompassing system-level architecture and processor micro-architecture. It provides a highly configurable simulation framework and supports cycle accurate emulation for most commercial instruction set architectures (ISA) and a broad range of CPU models. Since architecture simulators run orders of magnitudes slower than bare-metal, it allows configuration for trade-off between speed and accuracy. For example, it provides two system modes: the Full-System (FS) mode which simulates a bare-metal environment for running an OS like Linux and the much faster System-call Emulation (SE) mode where system calls are emulated. Figure 2.6 shows many ways to configure between simulation speed and its accuracy. Finally The framework also supports for power and energy modelling, so that we have more metrics than pure performance to evaluate our work.

2.4 Energy-Performance-Area Tradeoffs in Processor Architecture

Minimisation of power consumption while maintaining high performance in portable and battery powered embedded systems has become a major challenge for processor and system design today. While commodity processors are optimised for generic programs, but it does not execute arbitrary, especially domain specific, programs most efficient in terms of speed nor energy. Optimising a processor for energy efficiency needs to explore the tradeoffs in both micro-architectural design space and circuit design choices. In micro-architectural design space alone, the number of potential parameters is huge, with each having different effects on the overall performance. Table 2.1 shows the typical range for an Out-of-Order micro-architecture.

Parameter	**Typical Range**
Branch predictor	0-1024 entries
BTB	size 0-1024 entries
Cache levels	1-4 levels
Cache associativity	Direct-mapped(1 way) - Fully associative
I-cache size	2-32KB
D-cache size	4-64KB
L1 cache latency	1-3 cycles
L2 cache latency	8-64 cycles
DRAM latency	50-200 cycles
Fetch latency	1-3 cycles
Decode/Reg File/Rename lat.	1-3 cycles
Retire latency	1-3 cycles
Integer ALU latency	1-4 cycles
FP ALU latency	3-12 cycles
ROB size	4-32 entries
IW (centralized) size	2-32 entries
LSQ size	1-16 entries
Cycle Time	unrestricted

Table 2.1: Micro-architectural Design Space Parameters.

For different performance targets, a radically different processor macro-architecture may be most energy-efficient. Pareto-optimal trade-off curves in figure 2.7 shows energy-performance trade-offs of six macro-architectures of different execution paradigm and issue capabilities, which demonstrates the complicated relationship between the choice macro and micro architecture and their energy-performance trade-off; as the performance target is pushed, the optimal choice of macro-architecture changes to progressively more aggressive configurations.

Another important factor to consider is the die size. As the complexity of the processor grows, the number of transistors increases significantly, resulting in a increased manufacture cost and, especially for embedded systems, less portability. We will consider all these factors in our design space exploration in chapter 6.

2.5 McPAT Processor Modelling Framework

McPAT [19] is an architecture-independent integrated power, area and timing modelling framework which supports simulating a wide range of multicore and manycore processors configurations. It enables processor architects to perform design space exploration using metrics like the energy-delay-area product (EDAP) metric, achieving high performance, energy efficient and economic processor design. Figure 2.8 shows the structure of the McPAT framework,

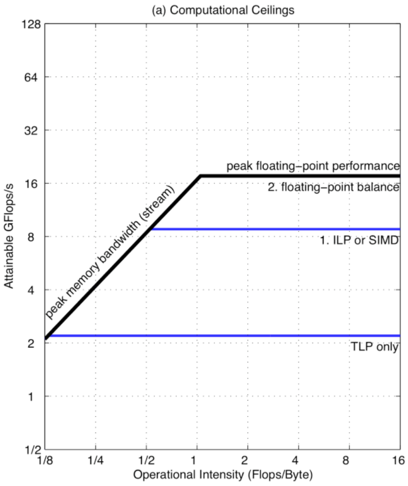
Power Modelling
: The system dissipate energy in several ways. Dynamic power is when the CMOS circuit charge and discharge the capacitive loads to switch between different states. Short-circuit power is caused by momentary direct circuit path between the source and the ground when transistors change state. They are both correlated to many parameters, the most significant ones being the supply voltage, clock frequency and activity factor. The first two factors are determined by system configuration and the last one is calculated from the cycle-by-cycle simulation. Since transistors have finite resistance, the electrons flowing through them causes leakage power, which increases as the number of active components increases.

Time Modelling
: Increasing frequency will increase the performance of the processor, however there is a physical limit to it since electrons takes time to travel across transistors. McPAT estimates the delay of the critical path and produces the maximum achievable clock frequency under certain lithography and supply voltage.

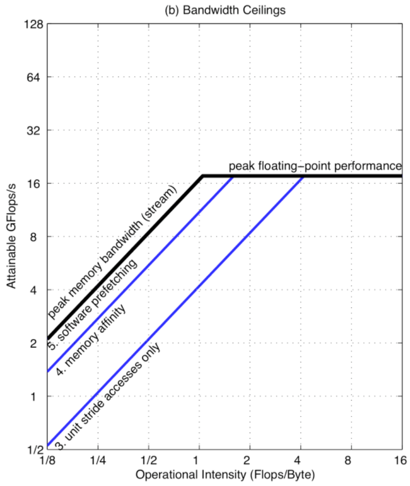
Area Modelling
: McPAT estimates the area of the silicon through an empirical method by fitting the configuration into a curve built from many existing processor

designs. Under unusual configurations, it may not be accurate and may not even be achievable due to the difficulty of wiring.

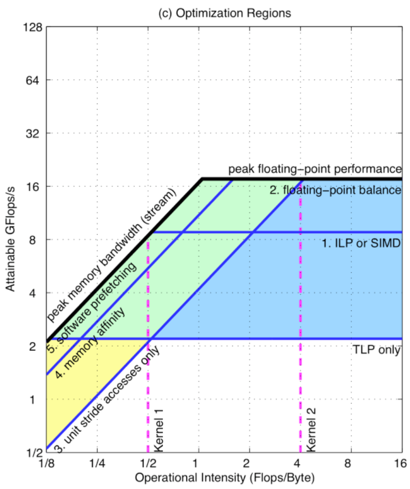
It accepts an XML interface for machine configuration and runtime log from a cycle-by-cycle system simulator like gem5, and produce an estimation of die area, dynamic energy usage and static energy leakage. It can be routed back to our simulator to refine our configurations. We will heavily use the framework in our design space exploration.



(a) Computational bottlenecks.

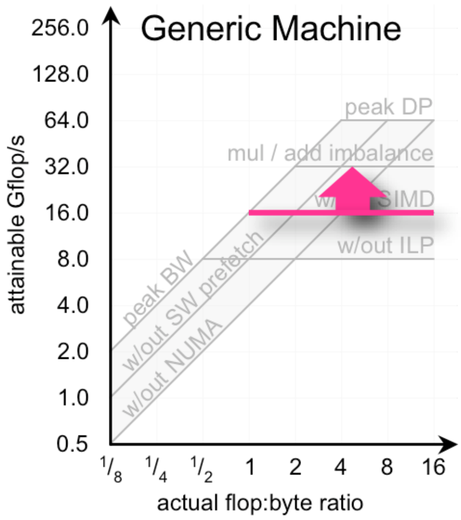


(b) Memory bandwidth bottlenecks.

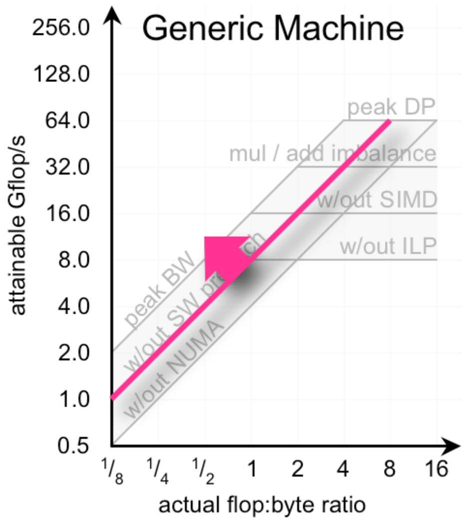


(c) Putting together.

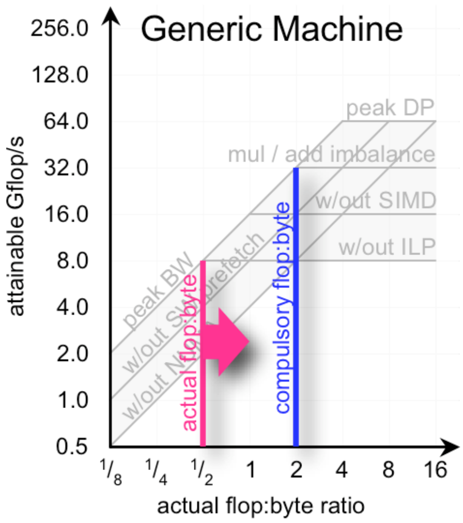
Figure 2.4: Roofline Model with Ceilings for Opteron X2. (diagrams are from [10])



(a) Maximising in-core performance.



(b) Maximising bandwidth.



(c) Minimising traffic.

Figure 2.5: General ways to optimise in the Roofline Model. (diagrams are from [10])

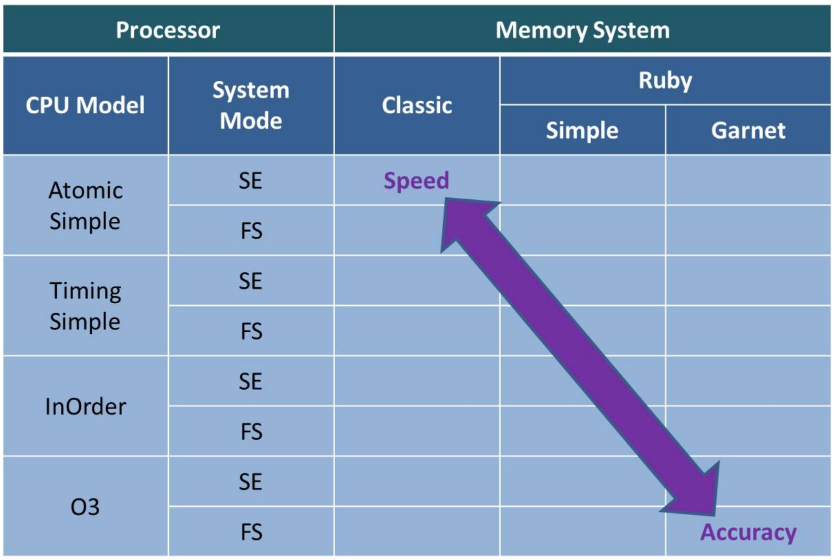


Figure 2.6: Speed vs. Accuracy Spectrum. (diagram is from [11])

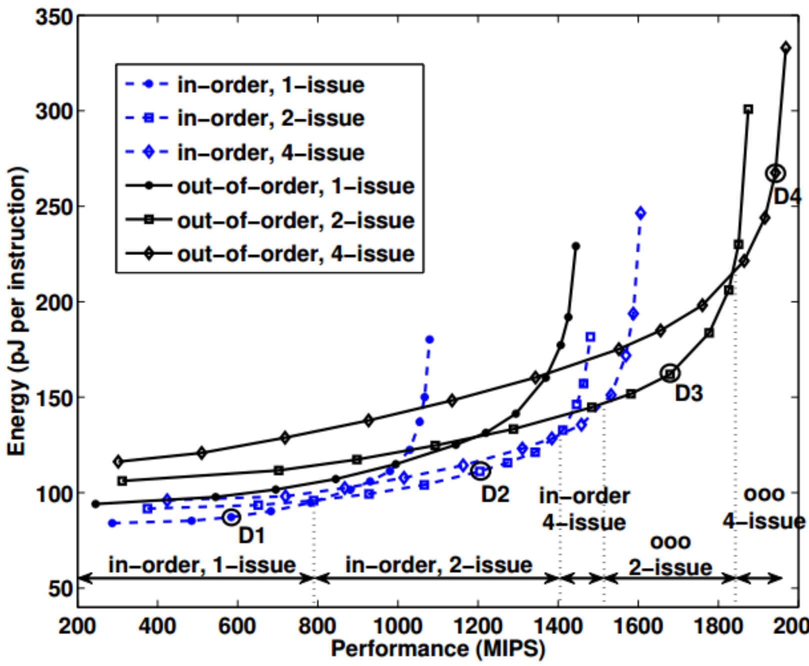


Figure 2.7: Energy-performance Pareto curve of six macro-architectures. (diagram from [18])

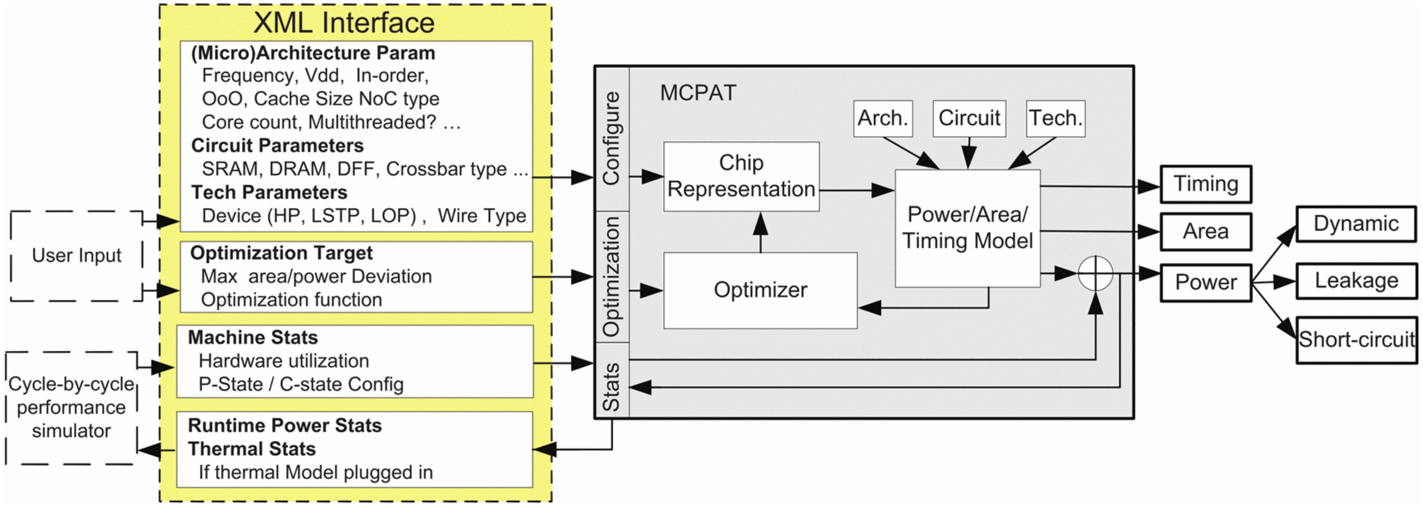


Figure 2.8: Block diagram of the McPAT framework. (diagram is from [19])

Chapter 3

Related Work

There are many works that give us inspiration and help us understand the progress on the related field. In this chapter we give an overview of these research. In section sec:sysanal we give an overview of several computer vision benchmarks and the analysis used to characterise their behaviour on embedded system. In section 3.2 we talk about existing work to use the Roofline model for analysing dynamic program behaviour. In section 3.3 we talk about a study of gem5 error margin on a similar platform as we are using.

3.1 Systematic Analysis for Computer Vision Kernels

Guthaus et al [20] introduced MiBench, a benchmark suite designed for embedded systems, in 2001. In the paper, the authors provided a systematic way to identify characteristics of each benchmark program in the MiBench including instruction distribution, branch predictability, and memory accesses using the SimpleScalar performance simulator.

By looking at instruction distribution, one can easily identify whether a program is computation intensive, control intensive or I/O intensive, and whether its fp/int instructions are balanced. The branch predication rates analysis gives clue about inherent branch correlation nature of the kernel. Finally the memory accesses characteristics shows the miss rate for different size and configurations of caches. All of these analysis are helpful for statically analyse kernels and understand their behaviour.

In MEVBench [21] paper, the authors looked into further profiling multicore scalability by introducing branch divergence and average multi-threaded instruction per cycle (IPC) analysis. These two metrics show how well the kernel may perform on heterogeneous design, where a power core will deal with singlethreaded applications plus additional small cores to support explicit parallelism. This inspires us to adding such configuration into our design space exploration.

3.2 Applying the Roofline Model on Hardware

The Roofline is a very simple model which only requires two factors about the hardware in order to develop the model, i.e. memory bandwidth and peak floating-point performance. However it has been used almost exclusively with back-of-the-envelope calculations , which is inaccurate and hard to work with for more complex algorithms such as FFT or dgemm. [22] Ofenbeck et al [22] presented a strategy on how to produce Roofline model plots with using performance counters on modern Intel platforms. Although it appears to be straight forward, there are many pitfalls around low level system details making the process complicated, followed are some of the aspects.

- The operating system has unpredictable effects on the kernel execution, introducing noise to the result. In case of context switches, the run time could bump by several times.
- The compiler may optimise by rewriting the code used to perform system performance benchmarking, results in over-estimating data.
- In order to collect correct results, some of the hardware functionalities need to be disabled, which deviates from common usage.
- The result still has a considerably large error margin, results will be hard to reproduce.

Another issue with using performance counters is that it is not portable across architectures, even different generations of the same architecture. Different architectures have drastically different collection of performance counters and for embedded systems, they can be inaccurate and does not convey enough run time information for plotting the Roofline model.

3.3 Sources of Error in Full-System Simulation

Gutierrez et al [23] studied using gem5 to simulate an ARM Versatile Express (Cortex-A15) development board and analysed the run time and key micro-architectural statistics error margin. The work provided a systematic method to precisely model an actual hardware relying on manufacturer manual. It shows that with appropriate modelling, the run time error for a mean percentage run time error of 5% for SPEC CPU 2006. This work establishes the foundation for our research into other ARM-based systems and helps us to establish confidence in our simulation results and allows us to consider the accuracy margin for our optimisation analysis.

Chapter 4

Kernel Analysis

We start our investigation by detailed analysis of the SLAMBench kernels. In section 4.1, by setting up the benchmarking environment and configure the gem5 simulator system to mimic the characteristics of our environment as closely as possible. We then attempt to construct the Roofline model for our simulation environment and add various optimisation ceilings to it.

In section 4.2, we perform a run time distribution analysis to find out the most time consuming kernels which will be the focus of our detailed analysis. Due to the slowdown effect of system emulation, we reduced and separated the selected kernels and picked a sample frame before detailed analysis using gem5. We first use the traditional methods to look at each specific runtime statistics to explain the performance gap and look for opportunities for optimisation.

In section 4.3, We plot the kernels onto the Roofline model and show that we can save a lot of human effort and obtain the exactly same results through this high level abstraction model.

4.1 Benchmarking Environment

Our tests are conducted on a Samsung Chromebook "Snow" running Cortex-A15 based Exynos 5 Dual (Exynos 5250). The system is chosen for its popularity in many commercial embedded systems of various form factors, ranging from single board computer and smart phones to desktop computers and tablets, and it is one of the most studied high performance embedded systems. It is also one of the recommended platforms for Cortex-A15 benchmarking by Linaro.

We set up the Chromebook using the environment shown in table 4.1. We simulated the system in gem5 with the best possible details using data derived from *ARM Processor Technical Reference Manual* (TRM) [24], *Exploring the design of the cortex-a15 processor* [25] and the Chromium project reference [26], with the guidance from *Sources of error in full-system simulation* paper [23]. Appendix A lists the changes we made to gem5 source code to accurately emulate the processor and the command-line arguments we use. Table 4.2 shows the comparison between parameters for the Samsung Chromebook and our gem5 configuration.

Parameter	**Setting**
Distro	Arch Linux ARM
Kernel	Linux 3.4.0-ARCH
Kernel Parameters	isolcpus=1 (for sequential tests)
cpufreq Driver	exynos_cpufreq
cpufreq Governor	Performance
Compiler	GCC 4.9.2
Compiler Flags	-O3

Table 4.1: System Configuration of Testbed.

Parameter	**Chromebook**	**gem5**
Pipeline	ARM Cortex-A15	O3CPU
CPU Clock	1.7GHz	1.7GHz
Branch Predictor	Bi-Mode	Bi-Mode
Fetch Buffer	16 Bytes	16 Bytes
FP Instr. Issue	2 per cycle	2 per cycle
FP Add Lat.	5 cycles	5 cycles
FP Mul Lat.	4 cycles	4 cycles
Cache Block	64 Bytes	64 Bytes
L1 I-cache	2-way 32KB	2-way 32KB
L1 D-cache	2-way 32KB	2-way 32KB
L2 cache	16-way 1MB	16-way 1MB
DRAM	DDR3L @ 800MHz	LPDDR3_1600_x32
Memory Size	2GB	2GB
System Bus	128 @ 500MHz	128 @ 500MHz

Table 4.2: Parameters for the Samsung Chromebook and our gem5 configuration.

With gem5, plotting the Roofline model is straight-forward, because both peak FLOPS and memory bandwidth are defined by our configuration. The system has 2 cores and each has 2 VFP/NEON engine. One engine is capable of executing 128 bit MAC operations and another only 64 bits, equating to 12 single position floating point operations per clock cycle. Therefore the theoretical ceiling for floating point operations per second is that multiplied by frequency (1.7GHz), which equates to 20.4 GFLOPS. Without balanced floating point operations, 6 single position is the theoretical limit (10.2 GFLOPS). Further without SIMD, only 3 operations can be executed at the same time (5.1 GFLOPS), and without ILP, only 2 single position operations are possible per clock cycle (3.4 GFLOPS). Finally with no task parallelism (TLP), only half of the number can be executed (1.7 GFLOPS).

The LPDDR3 module in Samsung Chromebook is capable of transfer 6.4 GB/s of memory per second, which can be found in `system.mem_ctrls.peakBW` simulation statistic. Without unit stride access, the memory access would be similar to random access, which is ranked at only 1.6 GB/s.

Figure 4.1 shows the theoretical performance and optimisation ceilings for our gem5 configuration.

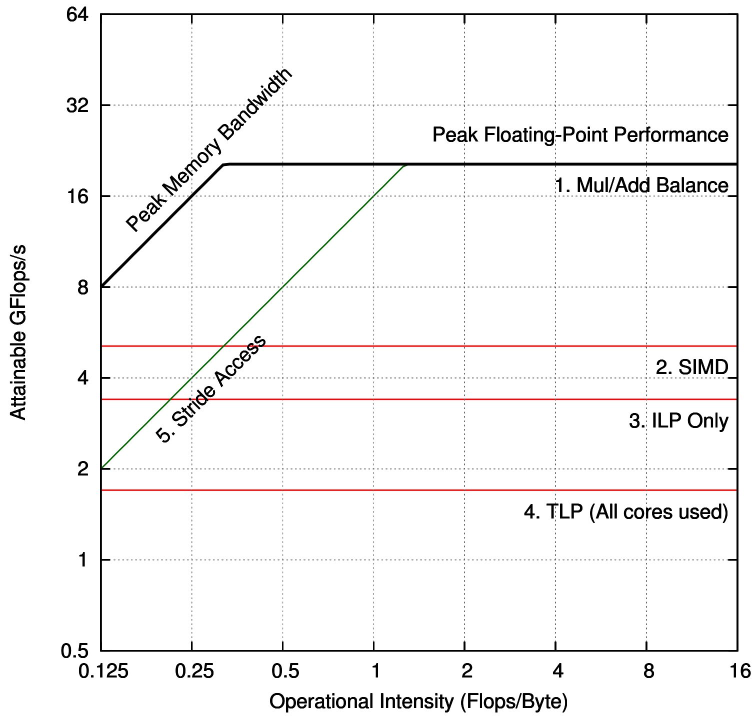


Figure 4.1: Roofline plot for gem5 simulation environment, produced using our analysis.

4.2 SLAMBench Kernels Analysis

The SLAMBench contains a number of kernels, each being independent from the others. Most of the kernels are invoked once for each frame, however some are called multiple times with different input sizes due to ICP algorithm and others are called every several frames, as explained in section 2.1.

To simplify our analysis, we tune the benchmark to execute all kernels for each frame. Due to the independent nature of the kernels, the analysis applies to other configurations as well. We attempted to lower the volume resolution down from $(256,256,256)$ to $(128,128,128)$, however we observe that this significantly changes memory behaviour and is no longer representative.

We ran the full benchmark for ICL-NUIM `'lr kt2'` dataset using `'cpp'` serial and OpenMP parallel implementations of SLAMBench. Figure 4.2 shows the run time distribution on ARM Cortex-A15, table C.1 and C.2 in appendix C includes detailed run time analysis for all kernels.

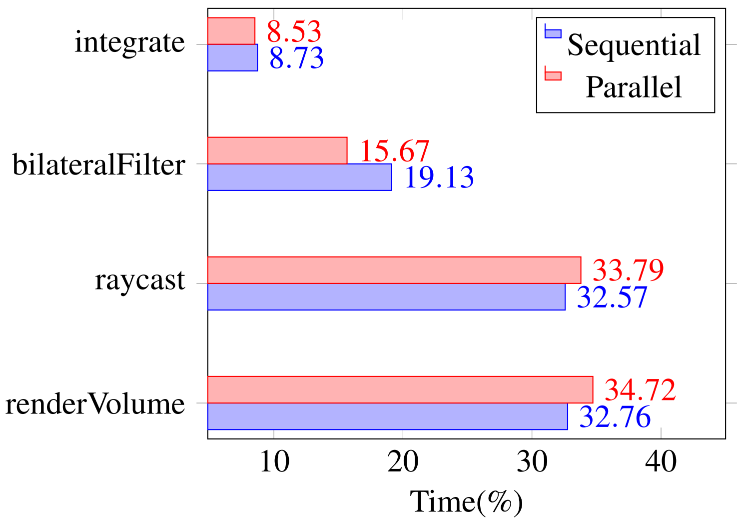
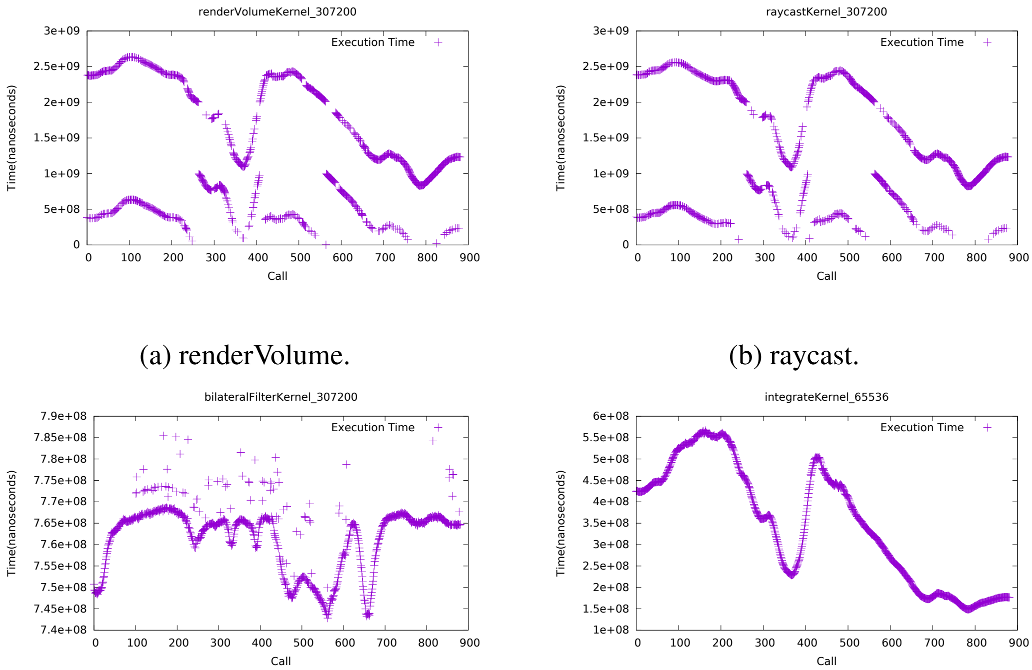


Figure 4.2: Execution Time of Selection of Kernels on Cortex-A15.

Since `renderVolume`, `raycast`, `bilateralFilter` and `integrate` kernels accounts for more than 90% of both serial and parallel run time, we will focus on these kernels for detailed analysis.

Input Dependency Analysis

We investigating their dependency on the input, since finding an representative frame could greatly reduce our benchmarking workload. We plot their run time across the entire 882 frames 4.3.



(a) renderVolume.

(b) raycast.

(c) bilateralFilter.

(d) integrate.

Figure 4.3: Timing of Selected Kernels across entire 882 frames.

Plots a and b are distinctively bi-modal. We took the experiment multiple times and found that the behaviour is non-deterministic. This anomaly does not reproduce on more powerful x86 machine. We do not fully understand this behaviour but we suspect this has to do with clock system call bugs in the ChromeOS kernel.

Plots a, b and d follow a clear trend. We then performed per frame execution path analysis of all 882 frames and found out that their run time is input dependent by that if the camera moves slower or moves into a previously mapped position, the raycast will be more likely to hit surface interface, reducing the number of times the main kernel loop needs to be executed. Knowing that, we decided to reduce the kernel by separating out the fifth frame run and put into test in the gem5 simulation environment, since it is both representative of the entire series by that it takes about average amount of time to complete and the volume no longer empty.

Finally, **bilateralFilter** (plot c) is purely dependent on the input. However since this kernel only uses camera input, which entirely fits into the L2 cache and is freshly used by mm2meters kernel, it is not interesting to examine closely. We will discard this kernel from our following detailed analysis as well.

We run the said simulation under gem5, and obtained their results (appendix D). The simulation on an modern X86 machine [1] takes around 1.5 hours to simulate a 1 second run on real hardware.

4.2.1 Dynamic Instruction Distribution

We start by looking into the dynamic distribution of instructions. There are four main classes of instructions: control logic(branches and jumps), integer calculations, floating point calculations and memory accesses (load and store). We can classify programs according to the percentage distribution of instruction classes.

- Control intensive programs are those with larger percentage of control logic instructions, which usually put heavy load on the branch predictor. For software optimisation, we are interested in techniques like loop unrolling to reduce the control overhead. As for hardware optimisation, a more aggressive branch prediction scheme and larger branch prediction table could benefit the prediction hits and a shallower pipeline could reduce mis-prediction penalty.

- Compute intensive programs are those have a majority of integer or floating point instructions. For software optimisation, we could balance the floating point and integer operations to utilise both kinds of functional units, or aggressively vectorisation to utilise instruction level parallelism (ILP). For hardware optimisation, we could configure the ratio of integer and floating point functional units to match the actual distribution, or to use a deeper functional unit pipeline to increase the overall throughput.

- Memory intensive programs are those which have larger proportion of memory access instructions. For software optimisation, we could tile the loop to increase memory access locality and reduce cache misses. For hardware optimisation, a larger cache could reduce cache miss rate, larger memory bandwidth and shorter memory access delay could reduce cache miss penalty.

We can easily extract the details from the gem5 execution report and figure 4.4 shows the dynamic instruction distribution for our kernels. We can immediately see that all three kernels are highly compute intensive, showing that we have greater need for **high computing throughput** than memory system.

[1] Intel i7-4770 CPU, 16GB DRAM

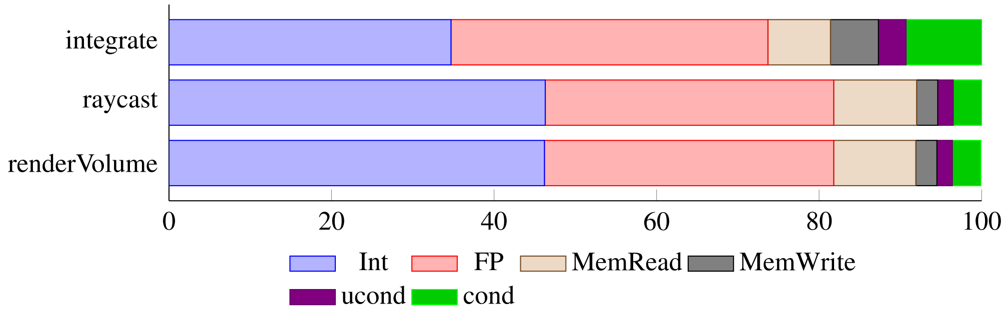


Figure 4.4: Dynamic Instruction Distribution for the gem5 simulation.

All of the kernels have very balanced integer / floating point instructions, which exploits both integer and floating point pipeline. This is confirmed by digging into the simulation log and found out that both pipelines are fully saturated, which indicates that we have **good balance in terms of the proportion of integer and floating point pipelines** for these two kernels, and **decoded instruction queue is large enough** to allow all the pending instructions to be issued.

By looking into the exact type of instructions (table 4.3), we found that in `renderVolume` and `raycast` kernel, although there are fairly balanced number of floating point addition and multiplication in both kernels, only around 10% are multiply–accumulate operations (MAC). This shows an opportunity for **floating point balance optimisation**.

Kernel	FPAdd	FPMult	FMAC
renderVol	34574330	48383886	9634641
raycast	35968831	49917533	11007828
integrate	28049207	65790	22476946

Table 4.3: Floating point balance of three kernels, integrate has very high utilisation of MAC unit while the others are seriously lacking.

The integrate kernel has less memory access and much more branch instructions. Detailed inspection shows very low L2 cache miss and branch prediction miss rate. It is because the kernel frequently access the depth which already resides in the L2 cache, and as we are integrating for every frame, we get far less movement than default thus branches are heavily biased, and the access to volume data is rarely accessed as well. In actual scenario, there will see a slightly higher L2 cache miss rate because it is only recently accessed and incurs compulsory misses.

4.2.2 Cache

Accesses to data in the cache hierarchy are magnitudes faster than the main memory system, and make a big difference in terms of average memory access delay. In table 4.4 we summarise the cache miss rate in our configuration. Although there seems to be a huge miss rate for the `integrate` kernel, there are actually few accesses to it, since everything fits into the L1 cache.

Kernel	dcache	icache	L2 Cache
renderVol	10.28%	0.00%	5.03%
raycast	10.36%	0.00%	5.29%
integrate	0.00%	0.00%	93.33%

Table 4.4: Cache miss rates of the kernels. Note that the cache hierarchy at the beginning of our simulation is warmed up, instead of potentially code in real world execution.

The L2 miss rate for `renderVol` and `raycast` is compulsory because they follow the search/stencil memory access pattern [7], and software optimisation cannot improve that much. However on the hardware size, addition of a **large L3 cache** could fit the entire volume and alleviate the memory pressure.

4.2.3 Branches

Another important thing to look at is the branches, especially predication rates. The Cortex-A15 utilises a bi-mode branch predictor [27] which is a dynamic prediction scheme, using two pattern history tables (PHT) indexed in gshare fashion, and a separate choice predictor, a table of 2-bit counters indexed in the lower bits of the branch address. The final prediction finally chooses one of the PHT based on the choice predictor table. Figure 4.5 illustrates the mechanism of the scheme. The scheme is highly efficient when the branch is heavily biased to one direction and less affected by destructive interference.

It turns out that the conditional branches in all of these kernels are heavily biased, resulting in a very low mispredication rate, as shown in table 4.5. This shows that compiler branch prediction hint is not likely to be effective and in architecture side we are are interested into using a **simple branch predictor** which might be a better trade-off between energy and performance.

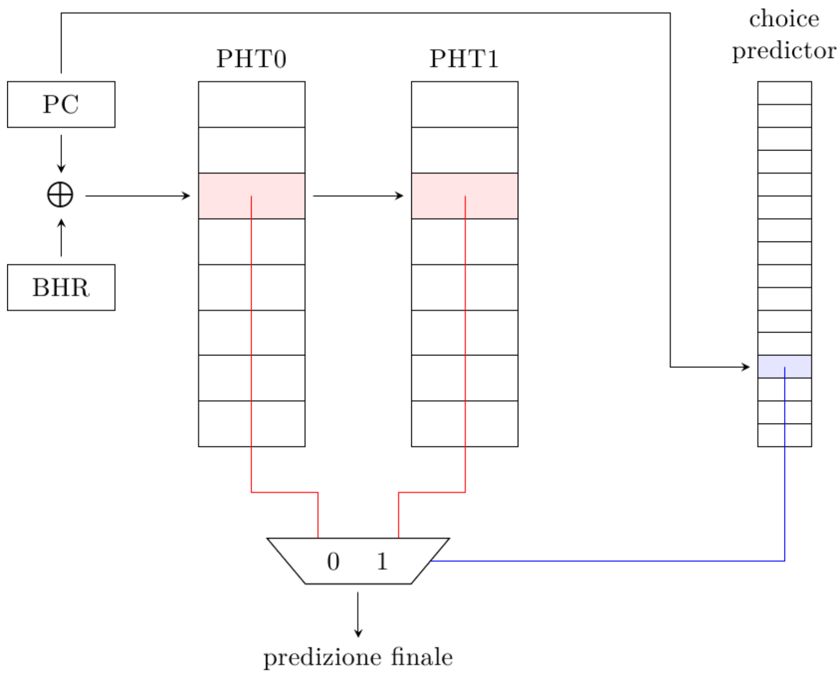


Figure 4.5: Structure of bi-mode branch predictor. (diagram is from [27])

Kernel	Branches(Million)	Mispredication rate
renderVolume	143.0	0.28%
raycast	142.1	0.28%
integrate	3.743	0.04%

Table 4.5: Branch prediction rate for three kernels.

4.2.4 Instruction Level Parallelism

Given the amount of power and die area costs to implement instruction level parallelism (ILP) on embedded systems, it is vital that we're fully utilising it to be energy and die area efficient. The number of instructions per cycle (IPC) is a good indication of the utilisation of ILP. Table 4.6 shows the IPC of the three kernels and we found that for `integration`, the compiler is already reordering most operations and high IPC is achieved. The other two kernels are not highly parallelised. By **reordering instructions** and **breaking dependency**, there could be more opportunity for the processor to execute instructions in concurrent. On the architecture side, a **larger instruction dispatch buffer** could benefit the performance and **shallower pipeline** coud reduce energy consumption.

Another thing to look at is the proportion of instructions that are SIMD (NEON)

Kernel	IPC
renderVolume	1.232
raycast	1.230
integrate	1.776

Table 4.6: Instructions per cycle for three kernels.

instructions, which could show how far the compiler auto-vectorised the kernel. However we cannot obtain that data such since the Cortex-A15 executes both floating point and SIMD instructions on the NEON engine and we cannot distinguish between them without runtime trace. Since this will further slow the simulation and we opt to not analyse it.

4.3 Roofline Model

Finally we plot the `renderVolume` and `raycast` kernels onto our previously established Roofline model. We omit integrate kernel since its image data mostly resides in L2 cache and there is rarely any access to the volumetric data. We calculate the operational intensity and floating operations per second (Flops) of the kernels, as shown in table 4.7, results figure 4.6. Both of the kernels have large operational intensity for our system, the memory bandwidth is hardly going to affecting the performance. However they only utilises around a quarter of peak floating point capability, and below ILP ceiling and floating point balance ceilings. This suggests we need to focus on improving ILP and reduce memory bandwidth on the architecture side to save power and die area. This simple model agrees with all our previous findings and proves to be really effective and labour saving method to look for optimisation opportunities. In addition, the model also shows that instructing SIMD instructions will help to improve performance while our previous analysis in section 4.2.4 fail to produce.

Kernel	Operational Intensity	GFLops
renderVolume	9.71	2.04
raycast	8.89	2.02

Table 4.7: Kernels roofline parameters under gem5 simulation environment.

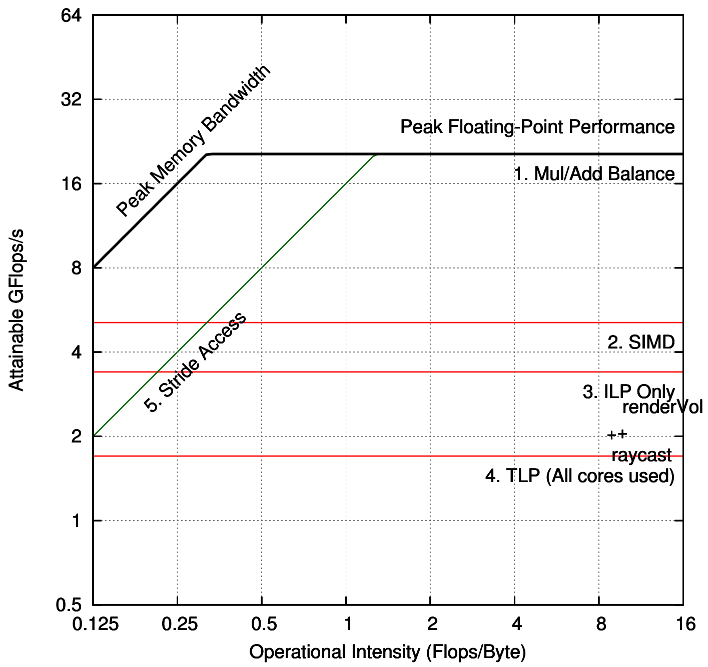


Figure 4.6: Roofline model with kernels plotted onto it. The machine rooflines are from figure 4.1, and kernels are plotted using our analysis in section 4.3

4.4 Summary

With our deep knowledge in the runtime behaviour of SLAMBench kernels, we are ready to improve the execution efficiency either by software optimisation or hardware optimisation. In chapter 5 we use the insights to use different software optimising technique to increase the throughput and in 6 we look into adapting the hardware for efficient software execution.

Chapter 5

Assessment of Optimisations

From our insights into the selected kernels, we are now able to perform guided optimisation. In section 5.1 we summerise the opportunities identified from previous chapter and in section 5.2 we implement the optimisation opportunities and finally in section 5.3 we evaluate our methodology in identifying and implementing optimisations in terms of achieved speed up and the amount of human labour.

5.1 Opportunities for Optimisation

We summarise the opportunities identified from our previous analysis.

1. All kernels can benefit from reordering instructions, unrolling loops and inlining functions to improve IPC.
2. `renderVolume` and `raycast` can benefit from vectorising and using SIMD instructions.
3. `renderVolume` and `raycast` can benefit from balancing floating point operations.

We determined that optimisation techniques to reduce memory bandwidth, e.g. software pre-fetching, unit stride access, are not likely to improve the performance. Since we are beyond memory bandwidth bound region, reducing memory traffic by better utilisation of cache is also not going to improve the throughput much.

5.2 Implementing Optimisations

5.2.1 ILP

We first attempted simplest and most common way which is by unrolling loops within kernels. The loops in these three kernels all operates on their respective input matrix thus are statically unrollable with the assumption that the size is power of 2. For illustration, we unroll the code in listing 1 by four times and results in listing 2. The performance benefit of this is modest, because the main kernel loop is very large in size and the benefit is partially cancelled out by the bloated binary size (more instructions need to be loaded into cache hierarchy). Benchmarks show merely 0.4% increase in performance across all kernels when unrolled by 2 times, and performance degradation when unrolled further.

```
for (y = 0; y < vol.size.y; y++) {
    for (x = 0; x < vol.size.x; x++) {
      // Main kernel loop
    }
}
```

Listing 1: Original main kernel loop.

```
for (y = 0; y < vol.size.y; y++) {
    for (x = 0; x < vol.size.x; x += 4) {
        // Main kernel loop 1
        // Main kernel loop 2
        // Main kernel loop 3
        // Main kernel loop 4
    }
}
```

Listing 2: Main kernel loop unrolled by four times.

Inlining function calls does not help the performance because the compiler is already doing so for smaller funtcions while the bloated binary size for inlining larger functions offsets the improvement in ILP.

5.2.2 SIMD

Because computer vision kernels operates on three dimensional space or three directions, and most operations are applied to all three dimensions simultaneously, which exposes great opportunity to vectorisation. We enabled `-ffast-math` and `-ftree-vectorize` to allow compiler auto-vectorisation and observed output of `-fopt-info-vec`. It turns out that the compiler was only able to vectorise a tiny fraction of the code, and most floating point matrix and vector operations are left unoptimised. However the use of `-fast-math` alone does significantly increases the performance due to reduced floating point accuracy and the benchmark shows a 13% percent increase in throughput.

Since all the kernels use CUTIL library for these computations, we hand vectorised the most used functions in it, so that every kernel would benefit. For example, the addition of `float3` type is vectorised by placing the three dimensions into the first three lanes of the 128-bit NEON registers, and apply `vsubq_f32` instruction on them, as shown in listing 3 and 4. We also modified `Volume::interp` to use the overloaded operation of `float3` in order to utilise vectorisation.

```
float3 operator+(float3 a, float3 b) {
    return make_float3(a.x + b.x, a.y + b.y, a.z + b.z);
}
```

Listing 3: Original `float3` addition.

```
float3 operator+(float3 a, float3 b) {
    float32x4_t vsum = vaddq_f32({a.x, a.y, a.z, 0},
                                 {b.x, b.y, b.z, 0});
    return make_float3(vsum[0], vsum[1], vsum[2]);
}
```

Listing 4: `float3` addition vectorised using `vsubq_f32`.

There is a significant overhead in this method because each time an operation is finished, it needs to be transformed from an `float32x4_t` type to `float3`, and the subsequent operation needs to transform it again. Thus the manual vectorisation actually slowed down the entire KinectFusion by 4%. We modified the codebase to transfer the results directly through the 128-bit type and the overall performance showed another 7% increase on top of the fast math optimisation.

5.2.3 Floating Point Balancing

Balancing floating-points is the one of the most tricky optimisation techniques. The imbalance is an inherent property of the kernel algorithm, and the compiler optimisation reorders the instructions and it is hard to understand from the output.

The work we can do is really limited. Listing 5 and 6 show one of the subtle optimisation we used. It used to take three operations (one multiplication, one division and one addition) and now it takes two (one division and one FMAC). Benchmarks show a small margin of improvement (0.3%) across three entire benchmark.

```
const float3 scaled_pos =
    make_float3((pos.x * size.x / dim.x) - 0.5f,
                (pos.y * size.y / dim.y) - 0.5f,
                (pos.z * size.z / dim.z) - 0.5f);
```

Listing 5: Original `scaled_pos` calculation.

```
const float3 scaled_pos =
    make_float3((pos.x / dim.x) * size.x - 0.5f,
                (pos.y / dim.y) * size.y - 0.5f,
                (pos.z / dim.z) * size.z - 0.5f);
```

Listing 6: `scaled_pos` optimised by balancing floating point operation. It now takes two cycles instead of three through the use of FMAC.

5.3 Evaluation

Manual fine-tuned optimisation is a complicated process, the programmer need to look at a large codebase and attempt various optimisation techniques trying to find a combination to reach higher performance. The gem5 analysis gives us important details about the runtime behaviour of the program and helps us to restrict the type of optimisation techniques to look at. The Roofline model, being a high level abstraction of program and machine behaviour, proves to be a more effective method which achieves similar conclusions as full system simulation while requiring far less low level details. It also gives an visual indication of how much machine capability is utilised.

Our optimisation effectively reduced the total runtime of the selected kernels by 21.3% and entire KinectFusion implementation by 20.7% on the Exynos Dual SoC. In the mean while, the human input is mostly focused in setting up the testing environment and implementing the optimisations, and spotting for optimisation opportunities is very effortless. Besides, the testing environment can be reused since it is not dependent on the program to be optimised. We conclude that our methodology for optimising computer vision kernels is a highly effective way in both the speed up achieved and the amount of effort required.

Chapter 6

Hardware Design-space Exploration

With our deep understanding of SLAMBench kernels, we finally look into optimisation hardware for more efficient execution of SLAMBench kernels using gem5. This is essentially a Pareto optimality problem where we are looking for a machine configuration with high performance with low energy consumption, as well as small die area so that the system is economic. We will only focus on the `renderVolume` kernel due to the massive slow down effect of full system simulation, but due to the generality of our methodology, this shall apply to any other computer vision kernels.

6.1 Methodology

As studied in section 2.4, the design space for processor architecture is huge, it is almost impossible to simulate every combination to find the optimal set up. However since we have performed bound-and-bottleneck analysis on our kernel, we can significantly reduce the search range. For our experiment, we will only study the `renderVolume` kernel due to limited time, however the methodology shall apply to arbitrary computer vision kernel due to its generality.

From our dynamic analysis, we found out that `renderVolume` is computational bound on our architecture, thus the main idea is to reduce the cost on memory and cache subsystem and focus on increasing its computing capability. The reduced range is shown in table 6.1.

We devise our exploration methodology based on the fact that in computer vision, it is important to achieve real time reconstruction of the 3D scenario of 30 frames per second. We reduce the energy-delay-area product (EDAP) metric from McPAT to:

$$\text{EAP} = \text{Energy} \times \text{Area}$$

Parameter	Typical Range	Current Setup	Reduced Range
Core count	1-32	2	2-32 [a]
Micro-architecture	OoO / in order	OoO	OoO / in order
Branch predictor	Simple / Complex	Complex (Bi-mode)	Simple / Complex [b]
Cache levels	1-4 levels	2	2-4 [c]
L1 Cache associativity	Direct-mapped(1 way) - Fully associative	2	1 - 2
I-cache size	2-32KB	32KB	2-32KB [d]
D-cache size	4-64KB	32KB	32KB
L2 cache size	256-4096KB	1MB	1-4MB [e]
L3 cache size	1-32MB	N/A	0-32MB
Memory bandwidth	1-128G/s	6.4GB/s	1-6.4GB/s

Table 6.1: Reducing design space parameters, using existing knowledge.

[a] More cores could significantly raise the computation capability.
[b] Loops are highly biased, a simpler predictor could save die space and energy.
[c] A large L3 cache can store the entire volumetric representation in cache.
[d] The size of our kernel is much smaller than 32KB.
[e] Stencil memory access pattern could benefits from larger context stored up in the cache hierarchy.

for our exploration so that we place equal importance to energy usage and processor manufacturing cost [1]. Our goal is thus to find the optimal configuration to minimise EAP. We integrated McPAT framework into gem5 through a script, by translating the gem5 statistics and machine configuration report to McPAT's XML input format. We devise a procedure briefly as follows:

1. For each parameter combination in the parameter set, we set up the gem5 simulator. The clock frequency is initially set to 1GHz.
2. Recompile the kernel targeting the selected architecture to allow the compiler to automatic optimise the code for the target architecture.
3. Simulate the hardware under gem5 simulator and determine the time to render.
4. Calculate the required clock frequency in order to achieve real time reconstruction.
5. Take the feedback and rerun the simulation under gem5, using the minimum required clock frequency.
6. Integrate the simulation results into McPAT model, then bisect the design space again if there is improvement in energy-area product.

6.2 Baseline System

Since the exact energy usage and die area is closely related to lithography technology, we need to set up a reference system for comparison. We first attempted the Exynos Dual set up, but it was not possible to reach real time reconstruction, because it requires a clock frequency of over 18GHz which is physically impossible to reach as determined by McPAT. We opt to increase the core count to 6 and were able to reach real time rendering at 6.3Ghz clock frequency McPAT reported $75.3\,\text{mm}^2$ in area and used $4.2\,\text{J}$ to render a single frame. The EAP for our baseline system is $75.3 \times 4.2 = 316\text{mm}^2\,\text{J}$.

6.3 Exploration

6.3.1 Micro-architecture

We start by looking into the micro-architecture. There are two main paradigms for CPU architecture, in-order and out-of-order (OoO) processors. In-order pro-

[1] The cost for manufacturing the processor is roughly proportional to die area.

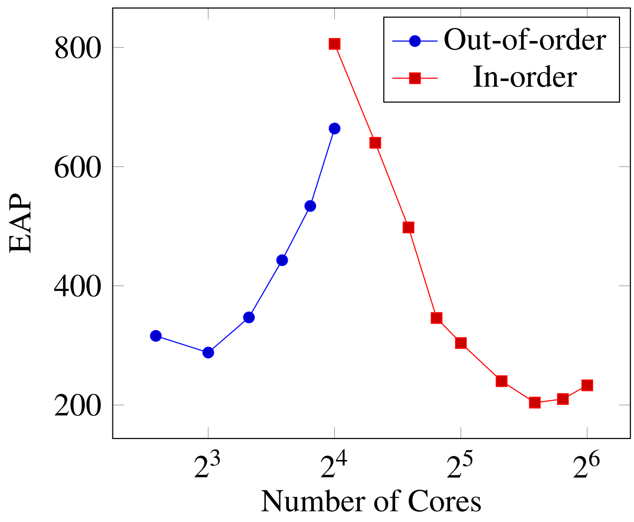


Figure 6.1: EPA for processors of different micro-architecture and number of cores.

cessors have a simple design, with a very shallow pipeline, very limited ILP. They are much smaller in die size, far less power hungry but a lot slower than OoO processors. Many small in-order processors are usually used to offset the low performance of a single processor. Another interesting component to look at is the branch predictor, there is a small margin of power and area saving for a simpler branch predictor.

We run the exploration on 4 to 16 out-of-order processors and 16 to 64 in-order processors using the procedure previously described. Figure 6.1 shows the trend of these configurations, and 48 in-order processors configuration is the clear winner with more than 35% reduction in EAP compared with baseline system and 29% lower than the best OoO. Figure 6.2 shows the EPA using either the complex bi-mode branch predictor, tournament predictor and the simplest 2-bit predictor on the optimal in-order processor. The tournament predictor has a marginally smaller EPA than the bi-mode predictor in Cortex-A15, but the 2-bit predictor shows worse results due to higher miss prediction rate offsetting the benefit from its hardware implementation.

6.3.2 Cache

We then attempt to change the cache parameters to further reduce the EPA. We have previously concluded that a larger cache is likely to benefit since it can store the entire volumetric representation. There are two possible methods of achieving this, a large high-latency L2 cache or a small L2 cache with a large L3

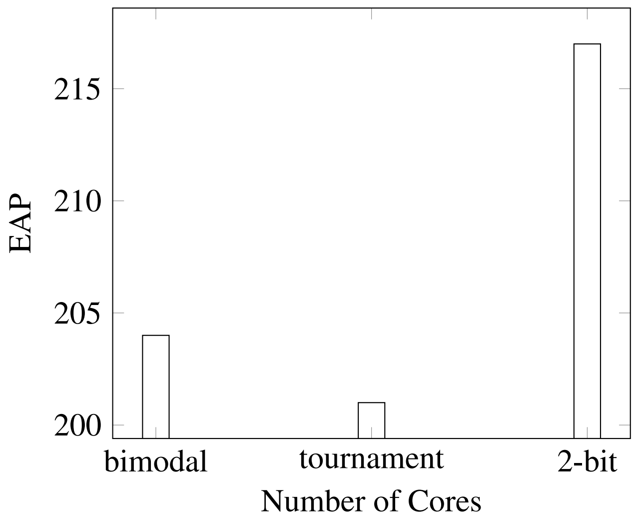


Figure 6.2: EPA for 48 cores in-order processors with various branch predictor.

cache. Since the size of the entire volume is 32MB, we will explore up to that limit. Figure 6.3 shows the effect of a larger 64-cycle latency L2 cache, the EAP improved by 7% compared to fast 1MB L2 cache. However the optimal is not at 32MB, matching the volumetric size, this is because with memory pre-fetching, a cache smaller than the entire object could still provide very high hit rate with a lower energy and die size usage. Adding an L3 cache is disappointing, as shown in figure 6.4. The EAP increases slightly, due to more components draining energy and taking up more die area.

Another conclusion we made is that a smaller L1i cache could save energy and die size because of the current under usage. We were not able to verify this as it triggers a bug within the gem5 simulator.

6.3.3 Memory Controller

With the processor micro-architecture, core number and cache hierarchy determined, we can finally decide the minimum memory bandwidth we require. Although it is not part of the original LPDDR3 standard, we can change its clock frequency in the simulator. However our experiments find out that with the *classic memory system* model we use in our simulation, it causes strange behaviours due to its inaccuracy. The *Ruby* model should be able to handle the modification but since it requires major changes to our benchmarking framework and time constraint, we will not investigate further.

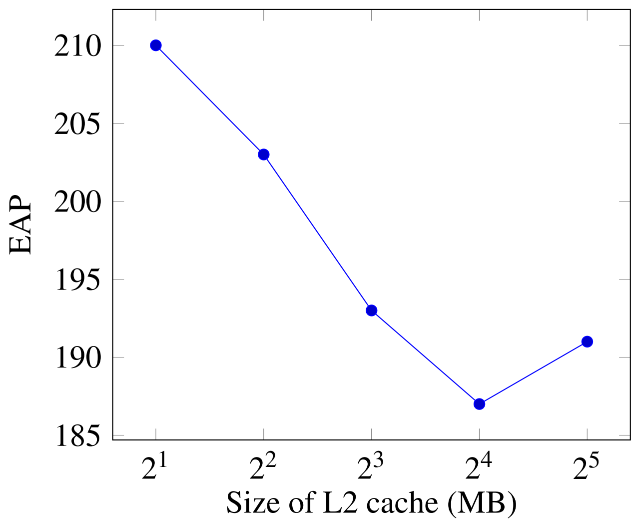


Figure 6.3: Effect of having a 64 cycle high latency L2 cache, 16MB is the optimal size which reduces EAP rating by 7%.

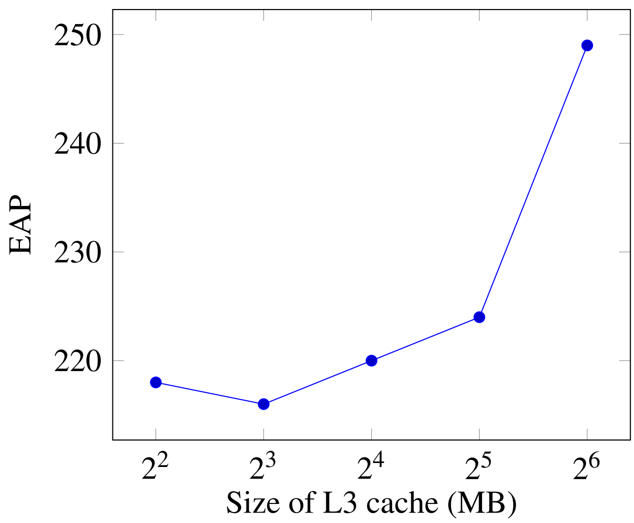


Figure 6.4: Effect of having a L3 cache, the rating decreases in this case.

6.4 Summary

Through detailed dynamic kernel analyses, we accurately identified the bottleneck parameters and significantly reduced the search space for hardware design. The results show a massively parallelled CPU design with 48 ”small” cores and a large slow L2 cache can reduce the EAP by massive 40% compared to the baseline system, through a much lower energy consumption compensating the increase in die size.

This is an interesting result in that our optimal configuration shows similarity with a GPGPU architecture, which both have large number of small cores and large memory bandwidth (in our case, since the volumetric size is small, we have a large L2 cache). This study shows a possibility to bridge the gap between CPU and GPU architecture. One of the possible impact is in heterogeneous CPU architecture, where we may be able to have a small number of ”large” cores to handle the sequential tasks and a large number of smaller cores to handle both simple sequential tasks and act as a GPU to reduce the cost of high performance SoC.

Chapter 7

Conclusions and Future Work

In this chapter, we give a summary of our findings and address the objectives set for the investigation. In section 7.2 we discuss some of limitation of the work and the direction for further investigation.

7.1 Summary

The aim of this project is to explore different implementations of the KinectFusion kernels and investigate where and how to introduce optimisations of various kinds, in order to improve its performance and reduce the energy usage. In chapter 4 we investigated the KinectFusion kernels and reduced our investigation to three kernels that takes up most execution time and performed detailed dynamic analysis for them. In chapter 5 we use analyses and model guided optimisation to improve the performance of the KinectFusion benchmark by 20.7% on the Exynos Dual SoC with small amount of manual effort. Lastly in chapter 6 we investigated optimising processor micro-architecture configurations to further improve performance and energy consumption and achieved 40% reduction in energy-area product (EPA).

In this thesis we have made the following contributions:

- We presented a detailed dynamic analysis on KinectFusion algorithm implementation on ARM Cortex-A15 architecture to understand its runtime behaviour, including dynamic instruction distribution, cache hierarchy, branching and instruction level parallelism. We plotted the kernels onto the Roofline model as an alternative approach.
- We demonstrated a systematic method to use the dynamic analysis and Roofline model to perform guided optimisation of SLAMBench kernels and

achieved 20% improvement in throughput and demostrated the effectiveness of the technique in terms of human effort.

- We devise a metric to assess hardware energy efficiency and die area and introduce a process to utilise dynamic analysis to greatly reduce hardware design exploration space. We reduced delay-area product by 40% through a GPGPU-like CPU design.

- From our conclusion that a GPGPU-like CPU design is energy efficient is efficient in processing parallelised programs, we discuss about the possibility to fuse CPU and GPU through an heterogeneous processor design.

7.2 Future Work

In this thesis, there are several areas where we do not have time to investigate and future work is required:

- Due to slowdown effect of system simulator, we were only able to investigate three kernels, two of which have very similar memory access behaviour. We need to extend the experiment to a more diverse set of kernels in order to prove the generality of the approach.

- We only used a fraction of the low level details provided by the system simulator in kernel analysis. Investigation in utilising the rest of the data for better understanding and using a higher level simulation system for faster turnaround time are two interesting areas to look at.

- We need to look at other optimisation methodology, manual or autotuning system, to compare and contrast in terms of speed up achieved and the amount of effort.

- We combined two simulation models (gem5 and McPAT) to study the system behaviour. This also multiplies the error range of the models from the actual hardware [23, 19]. Especially in the design space exploration where we used an unusual set of configurations of processor micro-architecture, work is required to investigate their accuracy under these extreme circumstances.

The success of using GPGPU-like many-core processor design for energy-efficient high-performance massively parallel workloads processing, which are usually thought to be only achievable by GPGPU is a important insight. Replacing GPU with massively parallel CPU cores can greatly reduce the silicon area has many applications including Internet of Things (IoT), extremely low-cost systems, accelerating OpenCL computation, etc. With many modern embedded systems already taking heterogeneous CPU design, this opens up a lot of possibility for future research.

Bibliography

[1] J. Aulinas, Y. R. Petillot, J. Salvi, and X. Lladó, "The slam problem: a survey.," in *CCIA*, pp. 363–371, Citeseer, 2008. 1

[2] "Dyson 360 website." `https://www.dyson360eye.com`. Accessed: June 2015. 1

[3] "Project tango website." `https://www.google.com/atap/project-tango`. Accessed: June 2015. 1

[4] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pp. 225–234, IEEE, 2007. 1

[5] R. A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pp. 127–136, IEEE, 2011. 1, 2, 3, 4, 5

[6] A. Geiger, J. Ziegler, and C. Stiller, "Stereoscan: Dense 3d reconstruction in real-time," in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pp. 963–968, IEEE, 2011. 1

[7] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167. 2, 26

[8] A. Handa, T. Whelan, J. McDonald, and A. J. Davison, "A benchmark for rgb-d visual odometry, 3d reconstruction and slam," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 1524–1531, IEEE, 2014. 2

[9] "Hololens website." `https://www.microsoft.com/microsoft-hololens/en-us/hardware`. Accessed: June 2015. 2

[10] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. 2, 3, 6, 7, 12, 13

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. 2, 3, 8, 14

[12] "Kinect website." `https://www.microsoft.com/en-us/kinectforwindows`. Accessed: June 2015. 4

[13] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on*, pp. 839–846, IEEE, 1998. 5

[14] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Robotics-DL tentative*, pp. 586–606, International Society for Optics and Photonics, 1992. 5

[15] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312, ACM, 1996. 5

[16] D. A. Patterson, "Latency lags bandwith," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004. 6

[17] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," 1995. 6

[18] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 26–36, ACM, 2010. 14

[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009. 10, 15, 43

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001. 16

[21] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 91–102, IEEE, 2011. 16

[22] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Puschel, "Applying the roofline model," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 76–85, IEEE, 2014. 17

[23] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 13–22, IEEE, 2014. 17, 20, 43

[24] A. ARM, "Cortex-a15 mpcore processor technical reference manual," 2013. 19

[25] T. Lanier, "Exploring the design of the cortex-a15 processor," 19

[26] "Samsung arm chromebook - the chromium projects." `https://http://www.chromium.org/chromium-os/developer-information-for-chrome-os-devices/samsung-arm-chromebook`. Accessed: June 2015. 19

[27] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 4–13, IEEE, 1997. 26, 27

Appendices

Appendix A

gem5 System Configuration

A.1 Modification to Source Code

In order to accurately emulate Samsung Exynos Dual (ARM Cortex-A15) SoC, we made the following changes to the gem5 source code.

The following diffs are relative to the tip of gem5 development branch.

A.1.1 L2 Cache Prefetch

```
--- a/configs/common/O3_ARM_v7a.py
+++ b/configs/common/O3_ARM_v7a.py
@@ -185,7 +185,7 @@
     size = '1MB'
     assoc = 16
     write_buffers = 8
-    prefetch_on_access = True
+    prefetch_on_access = False
     # Simple stride prefetcher
     prefetcher = StridePrefetcher(degree=8, latency = 1)
     tags = RandomRepl()
```

A.1.2 L1 I/DTLB Size

```
--- a/src/arch/arm/ArmTLB.py
+++ b/src/arch/arm/ArmTLB.py
@@ -63,7 +63,7 @@
     type = 'ArmTLB'
     cxx_class = 'ArmISA::TLB'
```

```
     cxx_header = "arch/arm/tlb.hh"
-    size = Param.Int(64, "TLB size")
+    size = Param.Int(32, "TLB size")
     walker = Param.ArmTableWalker(ArmTableWalker(), "HW Table walker")
     is_stage2 = Param.Bool(False, "Is this a stage 2 TLB?")
```

A.1.3 Fetch Buffer Size

```
--- a/src/cpu/o3/O3CPU.py
+++ b/src/cpu/o3/O3CPU.py
@@ -59,9 +59,9 @@
     iewToFetchDelay = Param.Cycles(1, "Issue/Execute/Writeback to fetch "
                                "delay")
     commitToFetchDelay = Param.Cycles(1, "Commit to fetch delay")
     fetchWidth = Param.Unsigned(8, "Fetch width")
-    fetchBufferSize = Param.Unsigned(16, "Fetch buffer size in bytes")
+    fetchBufferSize = Param.Unsigned(64, "Fetch buffer size in bytes")
     fetchQueueSize = Param.Unsigned(32, "Fetch queue size in micro-ops "
                                 "per-thread")

     renameToDecodeDelay = Param.Cycles(1, "Rename to decode delay")
```

A.1.4 2GB LPDDR3

```
--- a/src/mem/DRAMCtrl.py
+++ b/src/mem/DRAMCtrl.py
@@ -746,7 +746,7 @@
     dll = False

     # size of device
-    device_size = '512MB'
+    device_size = '2048MB'

     # 1x32 configuration, 1 device with a 32-bit interface
     device_bus_width = 32
```

A.1.5 Syscall Workaround

This workaround ignores timing system calls in order to appease the system when running unmodified SLAMBench kernels.

```diff
--- a/src/arch/arm/linux/process.cc
+++ b/src/arch/arm/linux/process.cc
@@ -381,10 +381,10 @@
     /* 259 */ SyscallDesc("timer_gettime", unimplementedFunc),
     /* 260 */ SyscallDesc("timer_getoverrun", unimplementedFunc),
     /* 261 */ SyscallDesc("timer_delete", unimplementedFunc),
-    /* 262 */ SyscallDesc("clock_settime", unimplementedFunc),
-    /* 263 */ SyscallDesc("clock_gettime", unimplementedFunc),
-    /* 264 */ SyscallDesc("clock_getres", unimplementedFunc),
-    /* 265 */ SyscallDesc("clock_nanosleep", unimplementedFunc),
+    /* 262 */ SyscallDesc("clock_settime", ignoreFunc),
+    /* 263 */ SyscallDesc("clock_gettime", ignoreFunc),
+    /* 264 */ SyscallDesc("clock_getres", ignoreFunc),
+    /* 265 */ SyscallDesc("clock_nanosleep", ignoreFunc),
     /* 266 */ SyscallDesc("statfs64", unimplementedFunc),
     /* 267 */ SyscallDesc("fstatfs64", unimplementedFunc),
     /* 268 */ SyscallDesc("tgkill", unimplementedFunc),
```

A.2 Command-line to Start Simulation

We opt to use System Emulation (SE) mode in our experiment for quicker simulation. The `arm_detailed` CPU is the `detailed` O3CPU model with certain parameters set to model a modern Out-of-Order ARM CPU.

Refer to section 4.2.

```
./build/ARM/gem5.opt configs/example/se.py \
  --cpu-type=arm_detailed --num-cpus=2 --cpu-clock=1.7GHz \
  --mem-type=LPDDR3_1600_x32 --mem-size=2GB \
  --caches --l2cache \
  --l1d_size=32kB --l1i_size=32kB --l1d_assoc=2 --l1i_assoc=2 \
  --l2_size=1MB --l2_assoc=16 \
  -c PROGRAM
```

Appendix B

Kernel Timing Trend

We show the per call kernel execution time graph for the rest of the kernels in SLAMBench in 4.2.

B.1 depth2vertexKernel

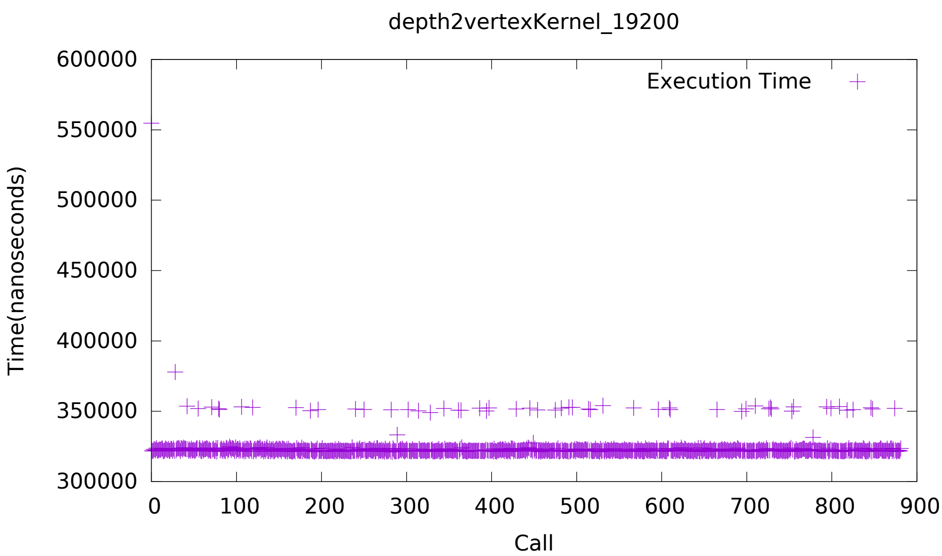


Figure B.1: depth2vertexKernel_19200

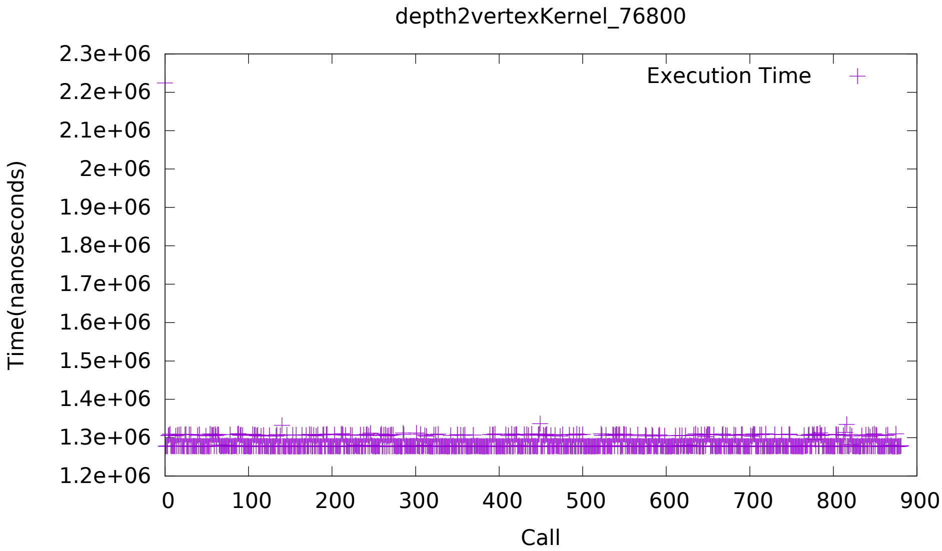


Figure B.2: depth2vertexKernel_76800

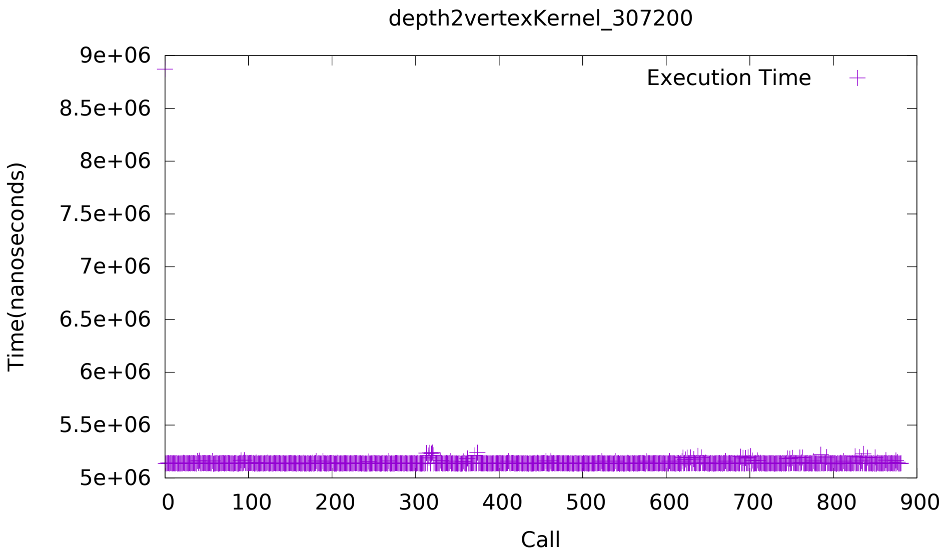


Figure B.3: depth2vertexKernel_307200

B.2 halfSampleRobustImageKernel

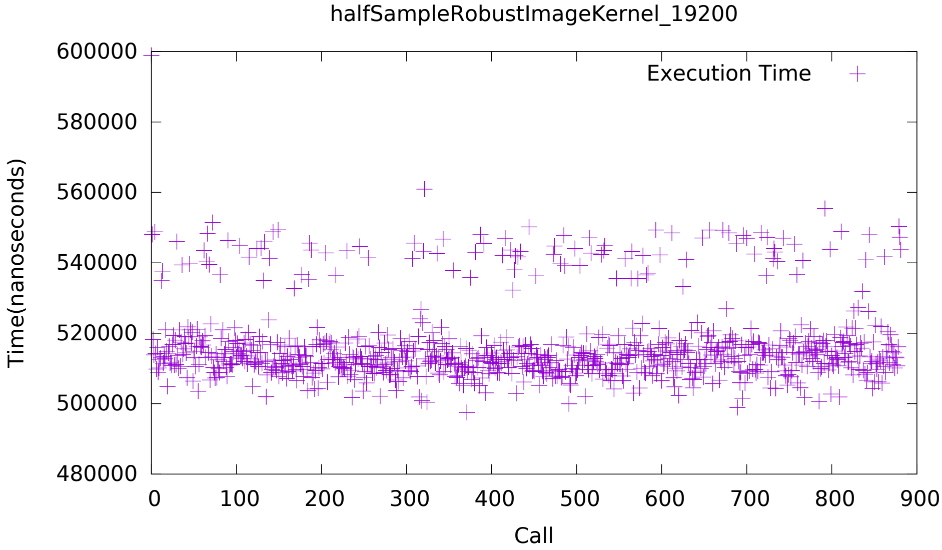


Figure B.4: halfSampleRobustImageKernel_19200

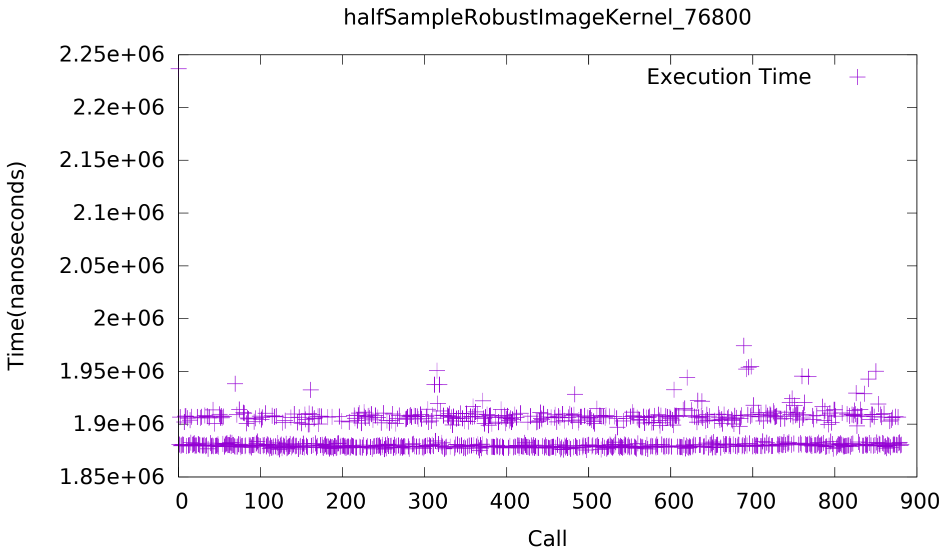


Figure B.5: halfSampleRobustImageKernel_76800

B.3 mm2metersKernel

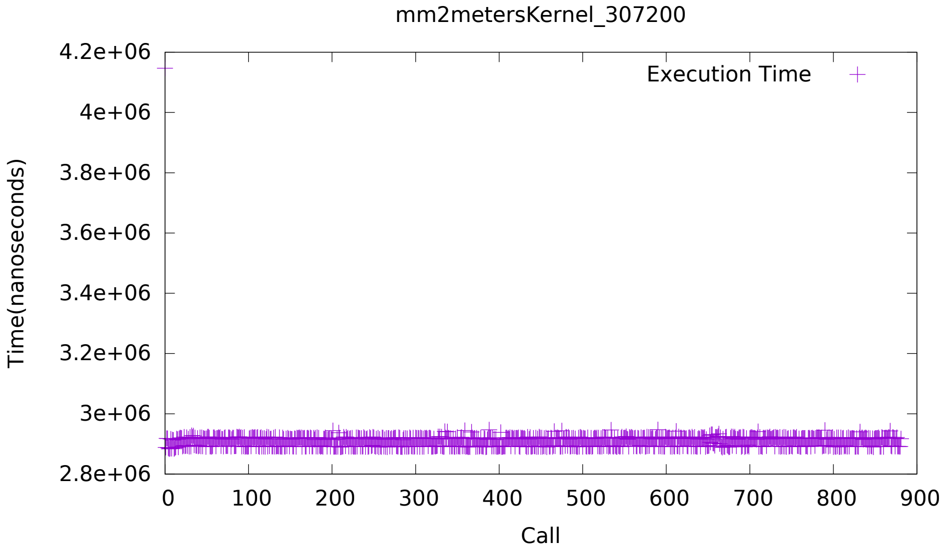


Figure B.6: mm2metersKernel_307200

B.4 reduceKernel

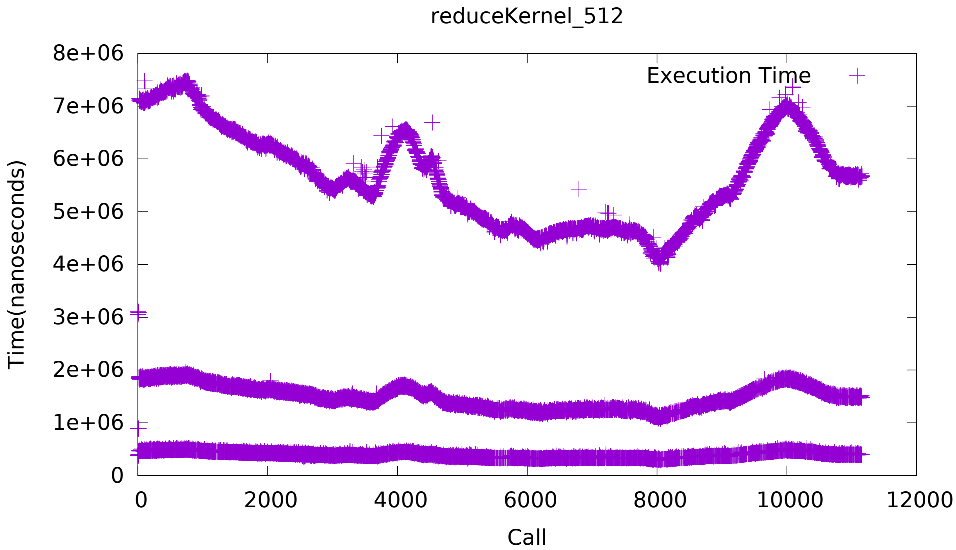


Figure B.7: reduceKernel_512

B.5 renderDepthKernel

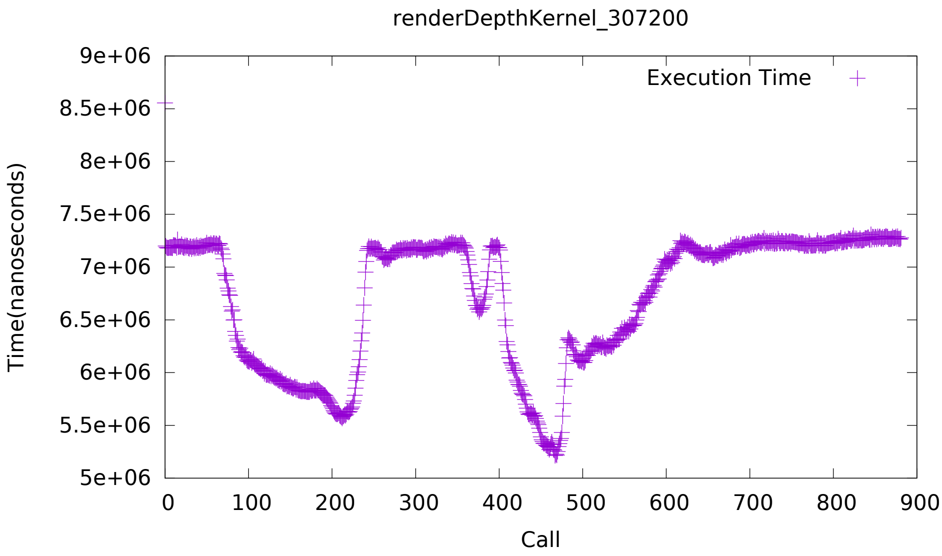


Figure B.8: renderDepthKernel_307200

B.6 renderTrackKernel

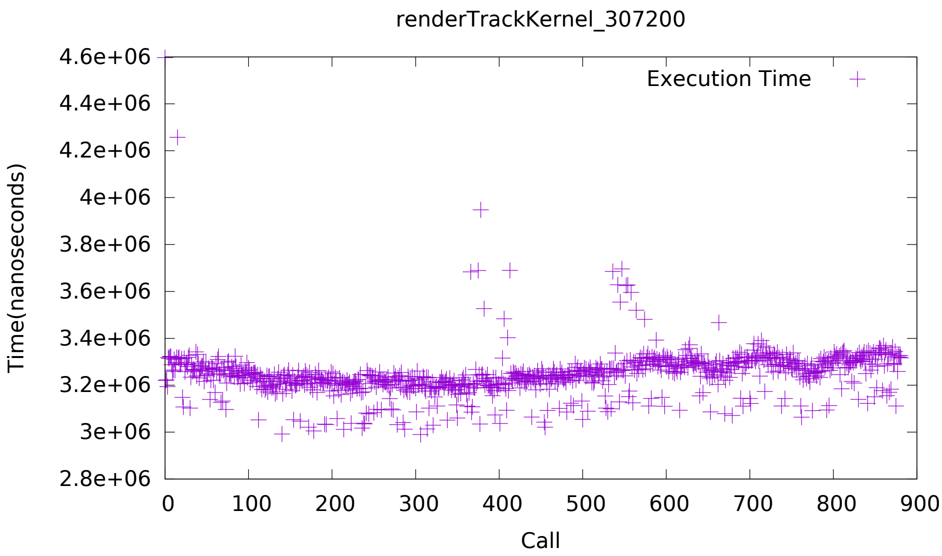


Figure B.9: renderTrackKernel_307200

B.7 trackKernel

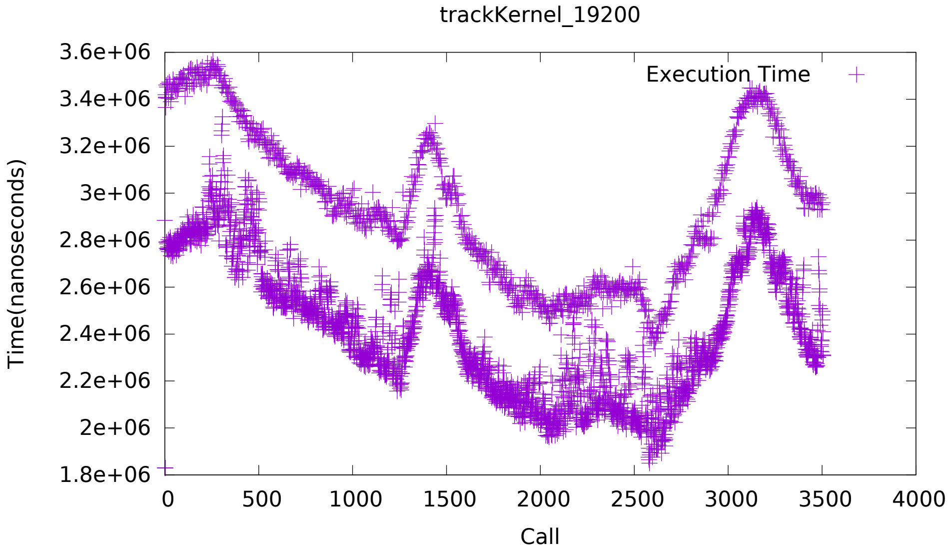


Figure B.10: trackKernel_19200

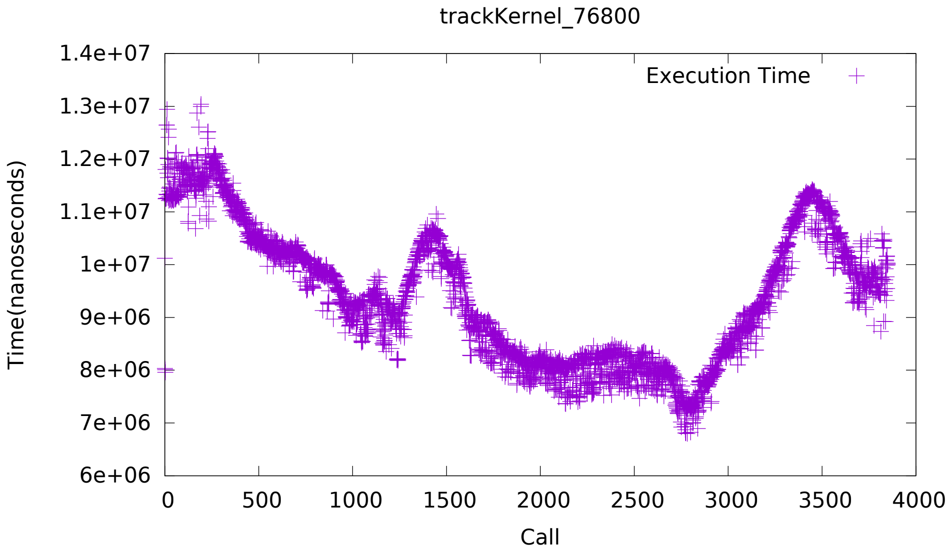


Figure B.11: trackKernel_76800

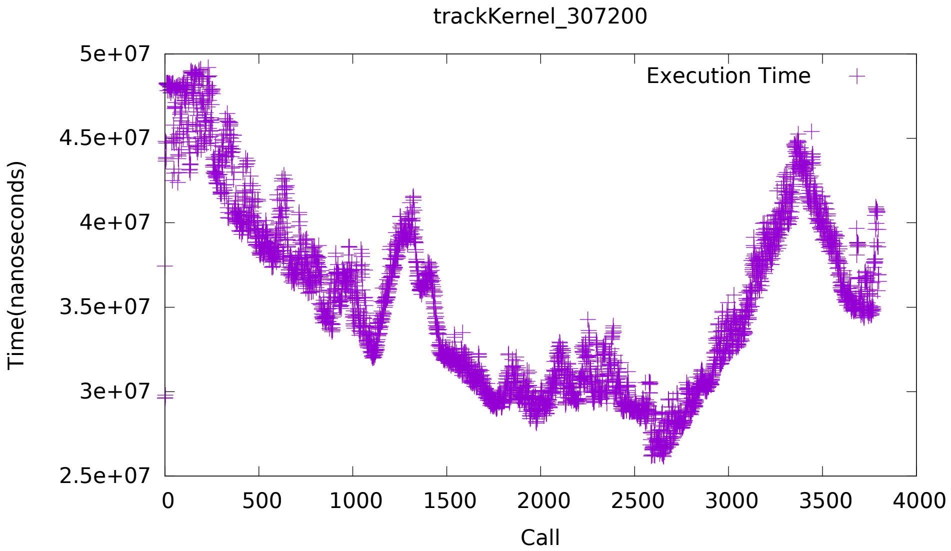


Figure B.12: trackKernel_307200

B.8 updatePoseKernel

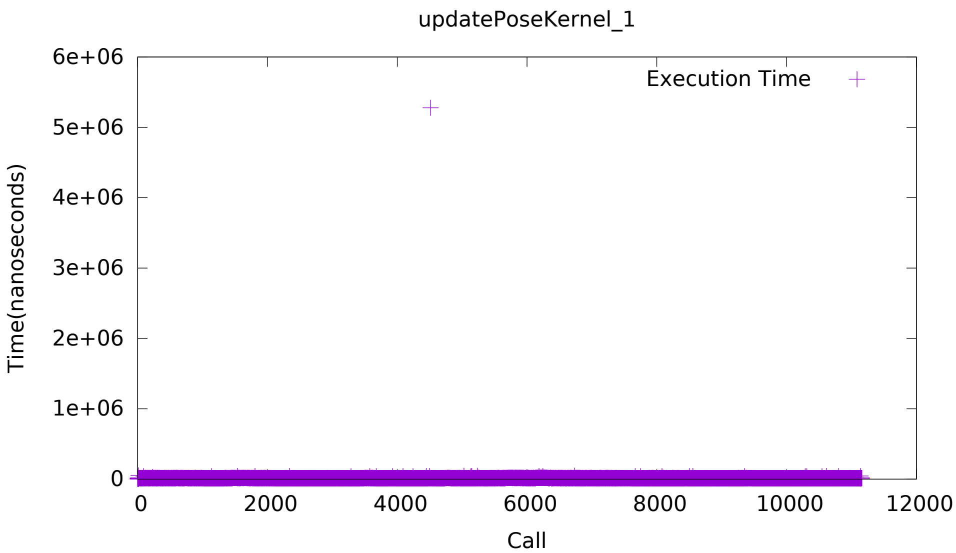


Figure B.13: updatePoseKernel_1

B.9 vertex2normalKernel

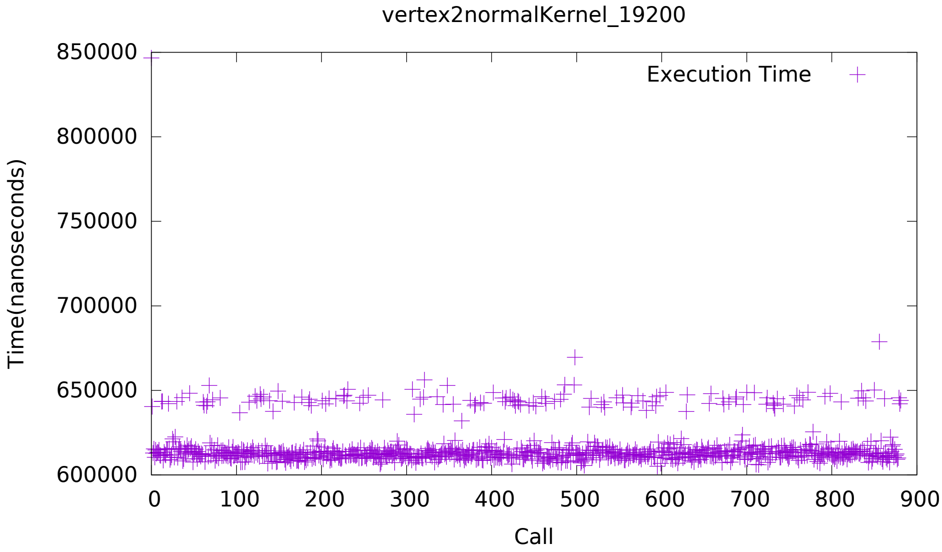


Figure B.14: vertex2normalKernel_19200

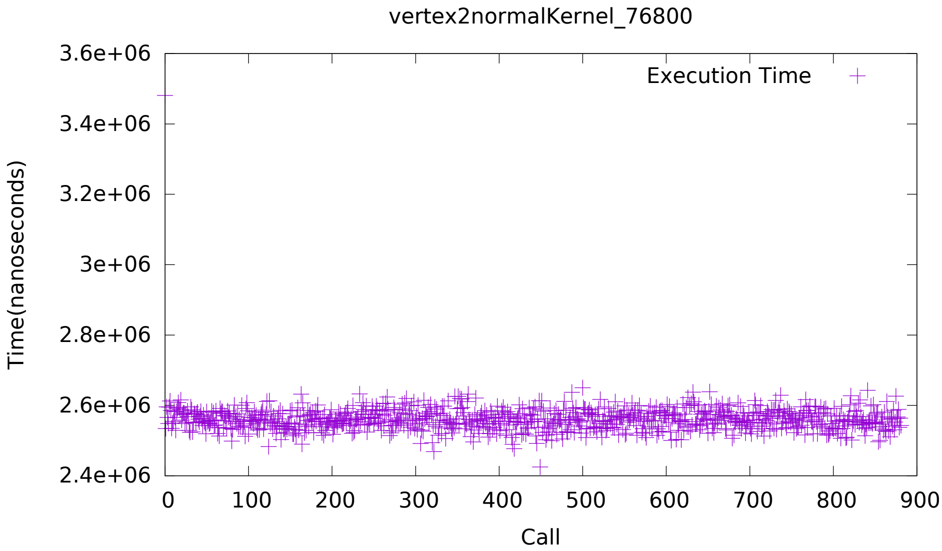


Figure B.15: vertex2normalKernel_76800

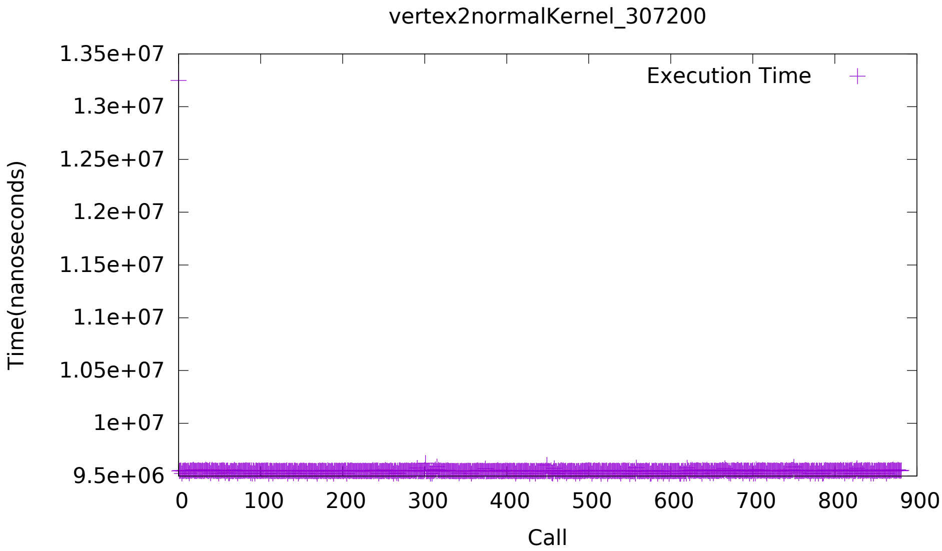


Figure B.16: vertex2normalKernel_307200

Appendix C

Sequential and OpenMP parallel runtime of untuned kernels

These are the run time statics for sequential and OpenMP versions of SLAM-Bench kernels running on the Samsung Chromebook, with the ICL-NUIM Living Room Dataset 'lr kt2' with 882 frames. % gives the percentage of time spent in the specific kernel running certain input size and Σ% gives the total time spent.

The system configurations are documented in section 4.2.

Kernel Name	Input Size	Calls	Total Time(s)	%	Σ%
bilateralFilter	307200	882	672.07	19.13	19.13
depth2vertex	19200	882	0.28	0.01	0.17
	76800	882	1.13	0.03	
	307200	882	4.53	0.13	
halfSampleRobustImg	19200	882	0.46	0.01	0.06
	76800	882	1.67	0.05	
integrate	65536	882	306.82	8.73	8.73
mm2meters	307200	882	2.57	0.07	0.07
raycast	307200	879	1144.39	32.57	32.57
reduce	512	11160	28.45	0.81	0.81
renderDepth	307200	882	5.93	0.17	0.17
renderTrack	307200	882	2.86	0.08	0.08
renderVolume	307200	882	1150.97	32.76	32.76
track	19200	3508	8.97	0.26	5.12
	76800	3851	13.46	1.03	
	307200	3801	36.19	3.83	
updatePose	1	11160	0.18	0.01	0.01
vertex2normal	19200	882	0.54	0.02	0.32
	76800	882	2.26	0.06	
	307200	882	8.43	0.24	

Table C.1: Sequential Execution Time on Samsung Chromebook.

Kernel Name	Input Size	Calls	Total Time(s)	%	Σ%
bilateralFilter	307200	882	354.98	15.67	15.67
depth2vertex	19200	882	0.18	0.01	0.16
	76800	882	0.67	0.12	
	307200	882	2.66	0.03	
halfSampleRobustImg	19200	882	0.36	0.01	0.07
	76800	882	1.33	0.06	
integrate	65536	882	193.35	8.53	8.53
mm2meters	307200	882	1.42	0.06	0.06
raycast	307200	879	765.52	33.79	33.79
reduce	512	11160	18.06	0.80	0.80
renderDepth	307200	882	3.46	0.15	0.15
renderTrack	307200	882	2.56	0.11	0.11
renderVolume	307200	882	786.67	34.72	34.72
track	19200	3508	7.02	0.31	5.63
	76800	3851	24.92	1.10	
	307200	3801	95.52	4.22	
updatePose	1	11160	0.18	0.01	0.01
vertex2normal	19200	882	0.33	0.01	0.27
	76800	882	1.34	0.06	
	307200	882	4.93	0.22	

Table C.2: Parallel Execution Time on Samsung Chromebook.

Appendix D

gem5 Statistics Reports

These are the runtime statistics obtained from gem5 simulator. The results are heavily reduced to fit into the report.

D.1 renderVolumeKernel

```
sim_seconds                                             1.371254
sim_ticks                                          1371253774884
final_tick                                         4595948169792
sim_freq                                           1000000000000
host_inst_rate                                            713682
host_op_rate                                              837834
host_tick_rate                                         193584854
host_mem_usage                                           2252708
host_seconds                                             7083.48
sim_insts                                             5055348923
sim_ops                                               5934776199
system.mem_ctrls.bytes_read::cpu.inst                       4224
system.mem_ctrls.bytes_read::cpu.data                  140844544
system.mem_ctrls.bytes_read::l2.prefetcher                989376
system.mem_ctrls.bytes_read::total                     141838144
system.mem_ctrls.bytes_inst_read::total                     4224
system.mem_ctrls.bytes_written::total                    1997632
system.mem_ctrls.bw_read::total                        103436830
system.mem_ctrls.bw_write::total                         1456792
system.mem_ctrls.bw_total::total                       104893623
system.cpu.fetch.icacheStallCycles                        402581
system.cpu.decode.BlockedCycles                       1150760890
```

system.cpu.rename.serializeStallCycles	0
system.cpu.branchPred.lookups	213696066
system.cpu.branchPred.condPredicted	138393488
system.cpu.branchPred.condIncorrect	394256
system.cpu.branchPred.BTBLookups	51741613
system.cpu.branchPred.BTBHits	51403963
system.cpu.branchPred.BTBHitPct	99.347430
system.cpu.commit.committedInsts	2873149991
system.cpu.commit.swp_count	0
system.cpu.commit.refs	499358375
system.cpu.commit.loads	397684551
system.cpu.commit.membars	0
system.cpu.commit.branches	209751945
system.cpu.commit.fp_insts	1396704528
system.cpu.commit.int_insts	1812656512
system.cpu.commit.function_calls	36405541
system.cpu.dcache.overall_mshr_miss_rate::total	0.102785
system.cpu.icache.overall_miss_rate::total	0.000000
system.l2.overall_mshr_miss_rate::cpu.inst	0.970588
system.l2.overall_mshr_miss_rate::cpu.data	0.050299
system.l2.overall_mshr_miss_rate::l2.prefetcher	inf
system.l2.overall_miss_rate::total	0.050301

D.2 raycastKernel

sim_seconds	1.366140
sim_ticks	1366140241116
final_tick	4587789179988
sim_freq	1000000000000
host_inst_rate	622077
host_op_rate	730626
host_tick_rate	168625308
host_mem_usage	2252708
host_seconds	8101.63
sim_insts	5039836469
sim_ops	5919263734
system.mem_ctrls.bytes_read::cpu.inst	4032
system.mem_ctrls.bytes_read::cpu.data	143906880
system.mem_ctrls.bytes_read::l2.prefetcher	4745536
system.mem_ctrls.bytes_read::total	148656448

system.mem_ctrls.bytes_inst_read::cpu.inst	4032
system.mem_ctrls.bytes_inst_read::total	4032
system.mem_ctrls.bytes_written::writebacks	7811456
system.mem_ctrls.bytes_written::total	7811456
system.mem_ctrls.bw_read::total	108814925
system.mem_ctrls.bw_write::total	5717902
system.mem_ctrls.bw_total::total	114532827
system.cpu.fetch.icacheStallCycles	391119
system.cpu.decode.BlockedCycles	1147977330
system.cpu.rename.BlockCycles	775429728
system.cpu.branchPred.lookups	212841453
system.cpu.branchPred.condPredicted	137264445
system.cpu.branchPred.condIncorrect	384332
system.cpu.branchPred.BTBLookups	51808916
system.cpu.branchPred.BTBHits	51537406
system.cpu.branchPred.BTBHitPct	99.475940
system.cpu.commit.committedInsts	2857637845
system.cpu.commit.swp_count	0
system.cpu.commit.refs	496321671
system.cpu.commit.loads	394410760
system.cpu.commit.membars	0
system.cpu.commit.branches	208935411
system.cpu.commit.fp_insts	1387089514
system.cpu.commit.int_insts	1808699100
system.cpu.commit.function_calls	36405541
system.cpu.dcache.overall_mshr_miss_rate::total	0.103591
system.cpu.icache.overall_miss_rate::total	0.000000
system.l2.overall_mshr_miss_rate::cpu.inst	0.984127
system.l2.overall_mshr_miss_rate::cpu.data	0.051233
system.l2.overall_mshr_miss_rate::l2.prefetcher	inf
system.l2.overall_mshr_miss_rate::total	0.052924

D.3 integrateKernel

sim_seconds	0.534651
sim_ticks	34651310368
final_tick	3773103562284
sim_freq	1000000000000
host_inst_rate	955324
host_op_rate	1189087

host_tick_rate	133021540
host_mem_usage	2252720
host_seconds	4019.28
sim_insts	3839719124
sim_ops	4779277103
system.mem_ctrls.bytes_read::cpu.inst	1408
system.mem_ctrls.bytes_read::cpu.data	448
system.mem_ctrls.bytes_read::total	1856
system.mem_ctrls.bytes_inst_read::cpu.inst	1408
system.mem_ctrls.bytes_inst_read::total	1408
system.mem_ctrls.bytes_written::writebacks	1920
system.mem_ctrls.bytes_written::total	1920
system.mem_ctrls.bw_read::total	3471
system.mem_ctrls.bw_write::total	3591
system.mem_ctrls.bw_total::total	7063
system.cpu.fetch.icacheStallCycles	67334
system.cpu.decode.BlockedCycles	224324688
system.cpu.rename.serializeStallCycles	0
system.cpu.commit.committedInsts	1614494049
system.cpu.commit.swp_count	0
system.cpu.commit.refs	270081579
system.cpu.commit.loads	152637718
system.cpu.commit.membars	0
system.cpu.commit.branches	251789827
system.cpu.commit.fp_insts	775106638
system.cpu.commit.int_insts	689973052
system.cpu.commit.function_calls	16777217
system.cpu.dcache.overall_mshr_miss_rate::total	0.000000
system.cpu.icache.overall_miss_rate::total	0.000000
system.l2.overall_mshr_miss_rate::cpu.inst	0.954545
system.l2.overall_mshr_miss_rate::cpu.data	0.875000
system.l2.overall_mshr_miss_rate::total	0.933333