Master's Thesis


Faculty of Electrical Engineering
Department of Measurement

# Energy efficiency of GPU applications in embedded systems

**Eduard Lavuš**

Supervisor: Ing. Sojka Michal Ph.D.
Field of study: Open Informatics
Subfield: Computer Engineering
Date: 2023-01-09

# MASTER'S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Lavuš Eduard** Personal ID number: **474497**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

## II. Master's thesis details

Master's thesis title in English:

**Energy efficiency of GPU applications in embedded systems**

Master's thesis title in Czech:

**Energetická efektivita GPU aplikací v embedded systémech**

Guidelines:

1. Become familiar with NXP i.MX8 QuadMax system-on-chip platform with dual GPU and Yocto Linux distribution for this platform.
2. Study the architecture and implementation (for NXP i.MX8) of the Linux graphics & compute stack (MESA, DRI, kernel drivers, OpenCL, Vulkan). Focus on features related to power management.
3. Update the Yocto distribution to allow switching between proprietary (galcore) and open source (etnaviv) GPU drivers. Also allow using GPU-based OpenCL platform together with CPU-based (PoCL) in the same application.
4. Develop a set of benchmarks to measure performance and power consumption of GPU compute (and optionally graphics) applications.
Use the benchmarks to evaluate energy efficiency and thermal properties of the available drivers and existing hardware and software power management features.
5. Document the results thoroughly. If you fix bugs or develop improvements in the graphics stack, submit them to the upstream projects.

Bibliography / sources:


- NXP: i.MX 8QuadMax Applications Processor Reference Manual, Rev. 0, 9/2021
- NXP: i.MX Graphics User's Guide
- Yocto project documentation: https://docs.yoctoproject.org/current/
- Bootlin: Understanding the Linux Graphics Stack: https://bootlin.com/doc/training/graphics/graphics-slides.pdf
- Sojka et al.: THERMAC deliverable 5.1: Benchmark suite and evaluation techniques


Name and workplace of master's thesis supervisor:

**Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC**

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **31.01.2022** Deadline for master's thesis submission: _____

Assignment valid until:
**by the end of summer semester 2022/2023**

| Ing. Michal Sojka, Ph.D.<br>Supervisor's signature | Head of department's signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean's signature |
|---|---|---|

# Energy efficiency of GPU applications in embedded systems

Eduard Lavuš

2023-01-09

# Acknowledgements

I would like to thank Ing. Sojka Michal Ph.D. for providing motivation, supervision and guidance without which this would not be possible. I would also like to thank the open-source community around Linux and the Mesa project for developing quality FOSS software upon which this work is based.


# Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

.....

# Abstract

As embedded hardware platforms get smaller and more powerful, heat production is a much bigger concern. In applications such as aerospace there are hard temperature constraints. We desire to optimize existing implementations to fulfill these constraints. Often, this solution is a combination of specific hardware, an operating system, libraries and the software application itself. In this thesis we explore configuration values of the hardware and operating system, and measure their effect on performance and maximum temperature. We perform our measurements based on previous work, and implement benchmarks to measure effects of our settings. We also design and implement software for automatically running measurements and configuring the system for reproducibility. We perform measurements on i.MX8 hardware from NXP, as part of an existing testbench. Finally we utilize gained knowledge to optimize an existing benchmark ADASMark. Our results show reduction of maximum temperature by almost 9% (over 1°C) while maintaining base performance. We also present how these configuration changes affect performance and temperature and can be utilized to find optimum for specific solution.



**Keywords**: embedded, OpenCL, Vulkan, Graphics API, Linux, GPU, power efficiency, heat output, maximum temperature



# Abstrakt

Vstavané hardwarové platformy sa stále zmenšujú a ich výkon sa zvyšuje. Preto nás často zaujíma aj ich tepelný výkon. V odvetviach ako letectvo a kosmonautika sú prísne tepelné požiadavky. Chceme optimalizovať existujúce implementácie aby požiadavky naplnili. Tieto riešenia sú často kombináciou špecifického hardwaru, operačného systému, knižníc a softwarovej aplikácie samotnej. V tejto práci skúmame konfiguračné hodnoty hardware a operačného systému, a meráme ich efekt na výkon a maximálnu teplotu. Merania zakladáme na predchádzajúcej práci a implementujeme evaluačné programy aby sme odmeraly efekty našich nastavení. Taktiež navrhujeme a implementujeme software pre automatické spúšťanie meraní a configurovanie systému pre reprodukovateľnosť. Merania spúšťame na hardwarovej platforme i.MX8 of NXP, ktorá je súčasťou existujúceho prípravku. Nakoniec použijeme nazbierané znalosti a optimalizujeme existujúci experiment ADASMark. Naše výsledky ukazujú redukciu maximálnej teploty o skoro 9% (viac ako 1°C) pri zachovaní pôvodného výkonu. Rovnako prezentujeme ako naše zmeny a nastavenia ovplyvňujú výkon a teplotu a je možné ich použiť pre optimalizovanie konkrétneho riešenia.



**Kľúčové slová**: vstavaný hardware, OpenCL, Vulkan, Linux, GPU, grafické API, energetická efektivita, tepelný výkon, maximána teplota

**Preklad názvu**: Energetická efektivita GPU aplikacií vo vstavaných systémoch

# Contents

# 1 Introduction

Computationally expensive tasks appear in most of todays engineering applications. The most common criterion in comparing applications is the time it takes to run a certain task to completion. We can optimize existing software and utilize newer, smaller and more powerful hardware to reduce this run time. However, heat output starts becoming an important concern as hardware shrinks. In addition to run time, we also have to consider whether the maximum temperature during computation remains under a certain threshold. This temperature depends on many factors, such as the ambient temperature, internal structure of the hardware and power consumption.

A core of a computer is composed of multiple components, such as the CPU, memory and memory controllers, and often a GPU. An important aspect of todays processing unit (CPU, GPU) is that it is not homogenous. The GPU is designed for specific workloads, especially for algorithms where the same operation is performed over a big amount of data. GPU is controlled by the CPU, which offloads suitable computations to it.

The goal of this work is to use capabilities available to software and their combination to either lower the maximum temperature or achieve better performance without increasing maximum temperature. The capabilities considered in this work are:

- clocking frequencies (dynamic voltage frequency scaling)
- code generation options (compiler optimization settings)
- disabling idle processing units (CPU/GPU cores)
- offloading part of work to different processing unit (e.g. 20% of the task is computed on the CPU, 80% on the GPU)
- selecting execution parameters optimized for used hardware (workgroup size, CPU affinity, memory access patterns)

To achieve this goal we will implement benchmarks and select a methodology. Then perform measurements on the NXP i.MX8 platform and verify results on an existing comprehensive benchmark ADASMark. The output of this work is an overview of how to achieve better performance at the same maximum temperature. The repository containing all source codes and relevant documentation is available online [1] and as a data disk appended to this work.

This work is part of the Thermac project (Fig. 1.1) which deals with reducing the operational temperature of computing platform [2].

## 1.1 Related work

Studying of relation between computational performance and temperature is not a new problem. Many works addressed that before us. Specifically, this work builds on the results by Hornof [3], who focuses on real-time safety-critical thermal-aware task scheduling. They propose an empirical model for estimating power consumption which affects heat output. In our work we focus primarily on GPU, on improving performance of a task given the same maximum temperature, thus this work is complimentary in the sense that combining these approaches can potentially have compounding benefits.

Very similarly Hosseinimotlagh and Kim [4] focus on scheduling workloads with known thermal behavior according to constraints set by the system to prevent unpredictability in real-time

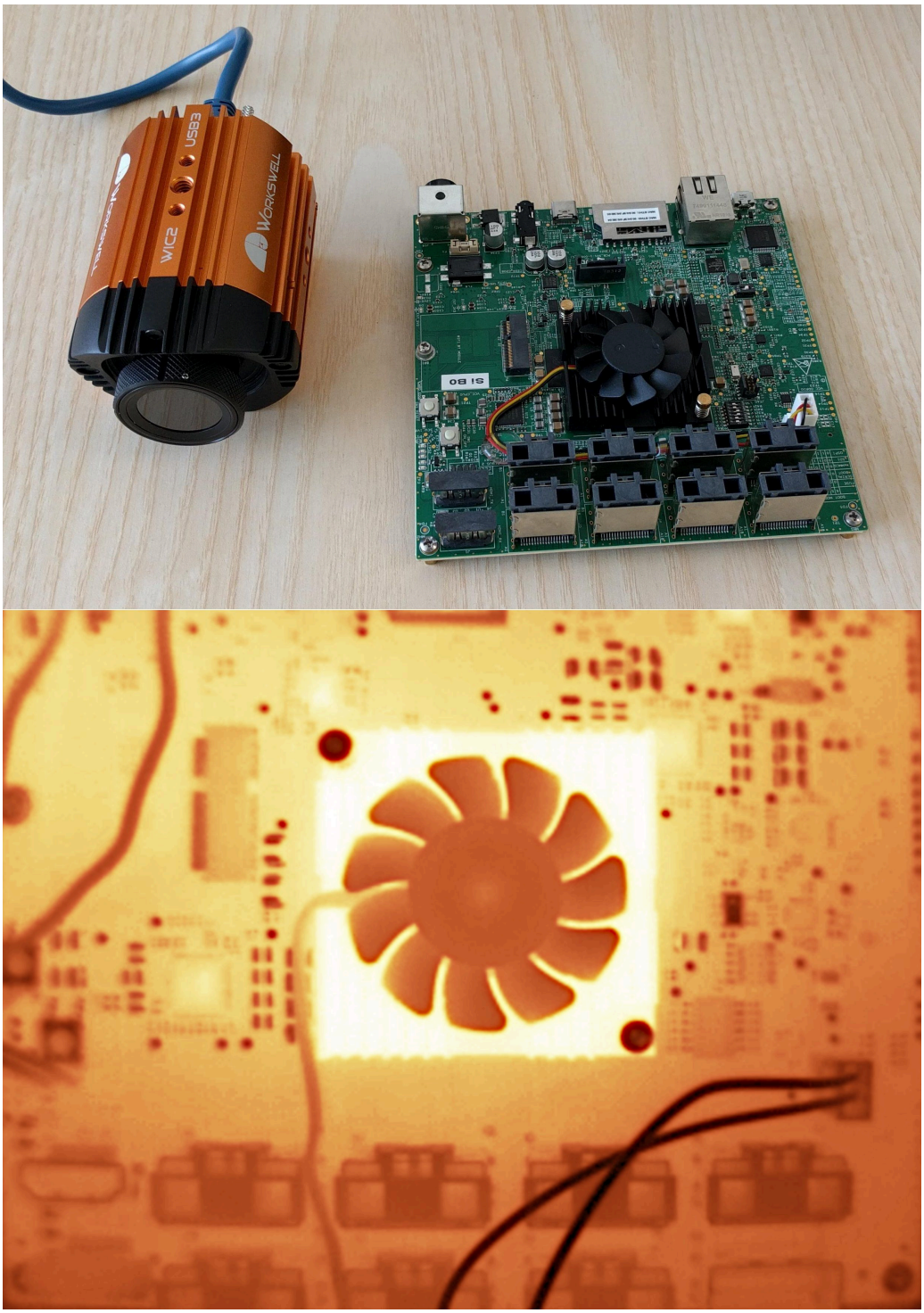


Figure 1.1: Thermac project focuses on measuring heat output and thermal efficiency.

systems. As such, it requires prior, usually experimental, knowledge of thermal behavior of given workloads. More tasks can be scheduled if we lower their maximum temperature, which is the focus of our work.

Another very interesting approach is seen in the work of Lee, Shin and Chwa [5] who focus on task scheduling improvements to avoid scheduling thermally intensive tasks on both the CPU and GPU at the same time, preventing constructive interference which is harmful to performance of both chips. This work focuses even more on CPU-GPU interaction. In our work we also care about CPU-GPU interaction but from the point of improving task performance.

Finally Lucas and Juurlink [6] focus on energy consumption based on GPU memory access, they do not measure power consumption nor heat output. Their results indicate that memory access patterns have effect on heat output through energy consumption and that memory controllers also produce non-trivial amount of heat. In our work we also have to account for memory access patterns and its effects in our empirical results. We however also focus on other processing units within the system.

# 2 Background

In this section we introduce the hardware platform, testbed and software tooling used to prepare and perform measurements.

We begin with an overview of hardware used, testbed preparation and later describe necessary low-level operating system and software knowledge. Finally we also mention where to obtain deeper knowledge about these parts.

## 2.1 System on a chip platform

The target hardware platform used in this work is the NXP i.MX 8QuadMax Multisensory Enablement Kit [7] board. This board contains an ARM Cortex CPU with two types of cores – 2x "big" Cortex-A72 cores clocked @ up to 1.6 GHz and 4x "LITTLE" Cortex-A53 @ up to 1.2 GHz. This is a relatively powerful CPU in the domain of mobile applications. The two big cores are akin to performance cores while the four LITTLE cores are geared more towards energy efficiency. These CPUs are designed for devices running complex compute tasks, including rich operating systems [8].

Second important part of this board is the Vivante Corporation GC7000XSVX GPU [9], which is comparable to ARM Mali GPUs commonly found in mobile phones. It is comprised of two subchips working together to achieve nearly twice the performance.

### 2.1.1 Board setup

The project hardware setup is shared with the earlier Thermac [2] project. This includes the i.MX8 board, controlling peripherals such as the turbot board for remote power control of the main board as well as additional sensors and a fan. This testbed is accessible remotely over an SSH connection to facilitate remote and automated measurements.

The booting solution is also reused from the project, making use of uboot [10], novaboot [11] and a stable server within the network to allow booting arbitrary versions of Linux over the network. Root filesystem is served over the same network using the unfs3 [12] implementation of an NFS server.

Fig. 2.1 provides an overview of the testbed.

## 2.2 Linux operating system

Linux as an operating system abstracts away many hardware specific details and provides uniform interface to configure the undelying hardware. In this work, we deal mainly with the graphics stack, so this section covers important parts of the graphics stack. Additionally, we cover necessary understanding for creating and interacting with a Linux system, such as the Yocto project, process scheduling and reverse-engineering.

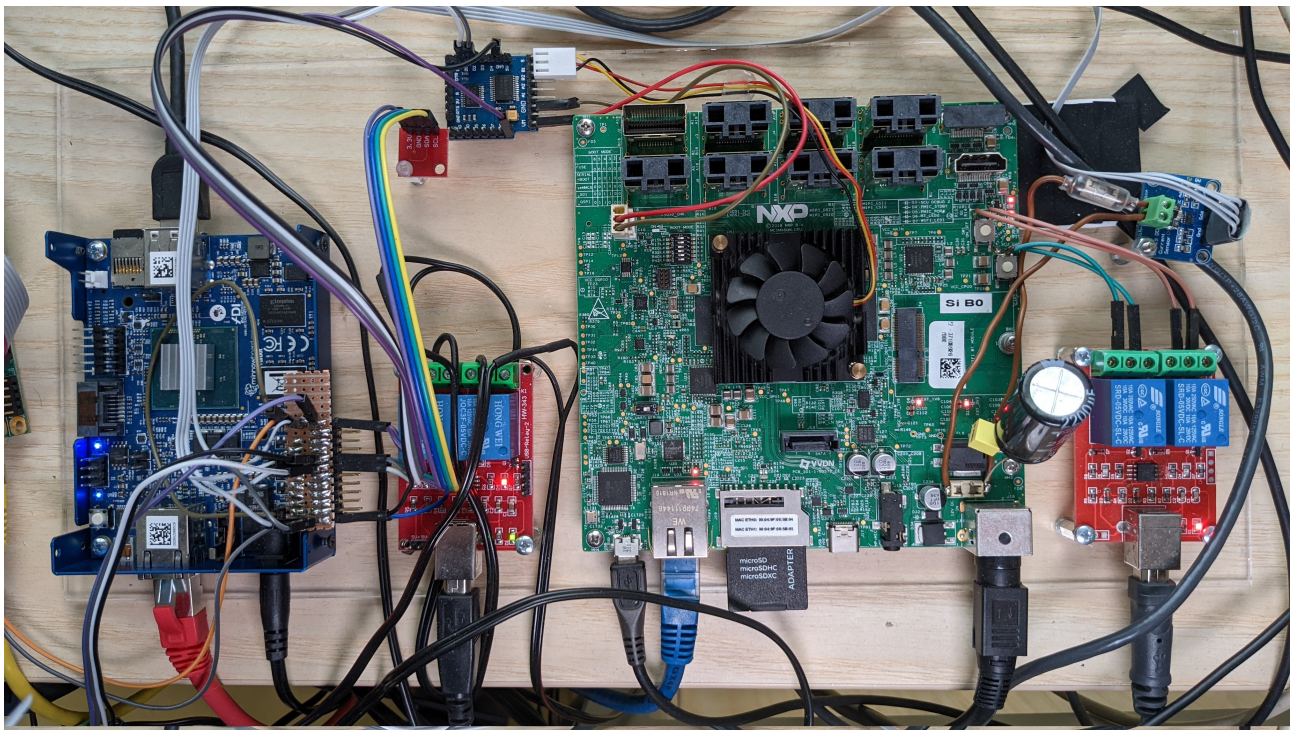


Figure 2.1: Thermac testbed overview. Green PCB in the center is the main iMX.8 board while auxiliary boards provide remote power management, fan control and sensor data

### 2.2.1 Process scheduling

To execute more threads than CPU cores the Linux kernel includes a scheduler which decides which execution thread should run at what time. Incorrect scheduler decisions may lead to unwanted variance of performance of specific workloads, as well as reduce performance. The Linux kernel includes multiple schedulers, and each can be further configured. The default scheduler, called the Completely Fair Scheduler (CFS), attempts to fairly distribute timeslices to running threads as they are required, with some configurability. Other schedulers are the realtime schedulers FIFO and Round-robin, which depening on priority allow a specific thread to always have a chance to run even at the cost of reduced system interactivity.

The Completely Fair Scheduler uses the so called “nice” value to determine how willing a process is to give up its resources (especially CPU time) to let other processes run. This value has no effect for realtime schedulers, which have their own priority settings always higher than the CFS priorities. For CFS the lower the niceness value the more resources the kernel will allow the process to consume before giving a chance to another process.

### 2.2.2 Hardware tuning

CPUs implement so called Dynamic Volatage and Frequency Scaling (DVFS), which allows to control its performance and power consumption. This is usually done automatically by the kernel and we did not tune this. However, individual cores of a CPU can also be hot-plugged – that is, turned on and off during the runtime of the system which is exposed in `/sys/devices/system/cpu/cpu{index}/online` pseudofile. When a core is turned off all processes are migrated out of it and processes are allowed to detect the current number of enabled cores.

Similarly the Vivante GPU supports dynamic voltage and frequency scaling (DVFS). Linux exposes interface to control it in `/sys/bus/platform/drivers/galcore/gpu_govern` pseud-

ofile. In case of the Vivante GPU available on our hardware platform this is exposed as three specific profiles (governors) `overdrive`, `nominal` and `underdrive` which can be set using the standard Linux interface. By default the GPU boots in `overdrive`.

### 2.2.3 Linux graphics stack

The Direct Rendering Manager (DRM) is a kernel module that gives direct hardware access to DRI (Direct Rendering Infrastructure) clients [13]. Direct Rendering Infrastructure is a framework for allowing direct access to graphics hardware, that is DRM is one way to integrate a device with DRI. The DRM facilitates communication between userspace (libdrm) and kernel (DRM) drivers, such as the ones provided by NXP and Mesa respectively. DRM module implements multi-client synchronization, which allows multiple userspace processes to simultaneously access the GPU, as well as a generic direct memory access (DMA) engine to streamline issuing commands to the GPU from these userspace drivers. The Fig. 2.2 provides a visual overview.

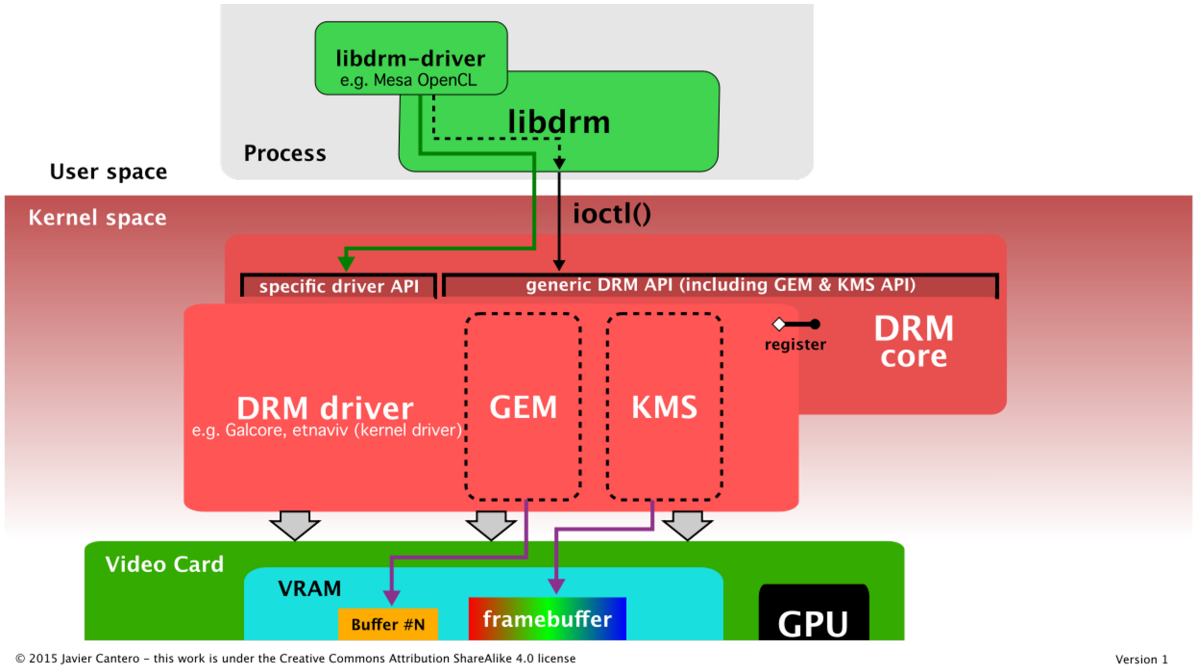


Figure 2.2: DRM Architecture overview [14]

On top of the DRM is the the Mesa project. It is an open-source implementation of OpenCL, OpenGL, Vulkan and other graphics and compute APIs [15], kernelspace drivers (often directly part of the kernel, but usually developed first as part of the Mesa project), as well as demo and utility tooling. Mesa is the de-facto standard in Linux ecosystem and has the widest support of API-driver combinations.

As a part of the Mesa project hardware drivers for the Vivante GPU lineup are available and are part of the NXP BSP yocto distribution, making this an important part of the graphics stack used. However, while the hardware driver implementation is part of Mesa, the userspace OpenCL and Vulkan implementations are still proprietary and provided by NXP.

## 2.3 OpenCL

OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms [16].

OpenCL provides an universal interface for execution of workloads in up to three dimensions on many different types of devices. Similar to classical programmable graphics pipeline and its shader programs, it uses kernel programs compiled either ahead-of-time or just-ahead-of-time (also known as online) to program the target device. It can also be used as an interface for any other kind of device capable of computation, such as a CPU.

Kernels are written in OpenCL C programming language, which is similar to the C programming language. This language is used as the standard interface for all targets and each target driver is tasked with providing its own compiler.

Since one hardware platform often contains multiple devices which can be targeted using OpenCL, and these devices often do not have the same manufacturer or driver developer OpenCL uses ICDs (installable client drivers) to register different drivers at startup. Then the user can enumerate these devices through the OpenCL API and select an appropriate device, or potentially instantiate multiple instances with different devices.

### 2.3.1 Vivante OpenCL driver

Vivante, the GPU manufacturer, provides drivers for its GPU lineup including this board. The supported OpenCL version is 1.2 with some partial support for OpenCL 3.0, as well as partial support of ICD loaders [17].

### 2.3.2 PoCL driver

PoCL (Portable Computing Language) [18] is an open-source OpenCL implementation which provides implementation of OpenCL 1.2 for CPU targets. With full support for ICD this can be used in addition to other OpenCL-capable devices on the hardware platform.

PoCL utilizes LLVM [19] library to compile OpenCL C to target platforms, leveraging its maturity and advanced optimizations, as well as many supported target platforms.

## 2.4 Vulkan

Vulkan is a low-overhead, cross-platform graphics API targeting high-performance 3D graphics applications [20]. Vulkan is a successor to the OpenGL API as well as incorporating parts of the OpenCL API with the aim to eventually converge and allow seamless transition from OpenCL to Vulkan applications. Vulkan supports compute shaders (kernels in OpenCL) on its own, however the official Vulkan shader bytecode is SPIR-V, and drivers are not required to consume source-code directly, only the SPIR-V bytecode.

### 2.4.1 Vivante Vulkan driver

Vivante provides Vulkan 1.2 drivers for this board, similar to OpenCL libraries. These drivers are distributed alongside the NXP Yocto layer and are installed by default.

### 2.4.2 Vulkayes library

Vulkayes [21] is a wrapper library for Vulkan API for the Rust programming language [22]. It is built on top of the ash library which provides C ABI bindings to the system Vulkan implementation. This library provides additional type safety while having minimal runtime cost. Vulkayes was developed by the author of this thesis [23].

## 2.5 Yocto project

The Yocto Project [24] is an open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture. It is a tool to describe a recipe (set of steps to perform, scripts to run) for a reproducible build of a complete Linux distribution, including patches, custom packages and build-time scripting. An overview can be seen in Fig. 2.3.

The basis of the yocto project is usage of layers. A layer consists of configuration and recipes for specific packages. Each successive layer has full control over the previous layers and can expand, mutate or disable any recipe of a previous layer. This allows vendors to fully customize their software solution while building on top of a common core, sharing benefits like upgradability, support for external packages and more.

For example, the very base layer is the OpenEmbedded layer which provides the very base Linux kernel and packages. On top of that, there might be layers for specific projects, such as a layer for clang, or for specific hardware, such as the one provided by NXP.

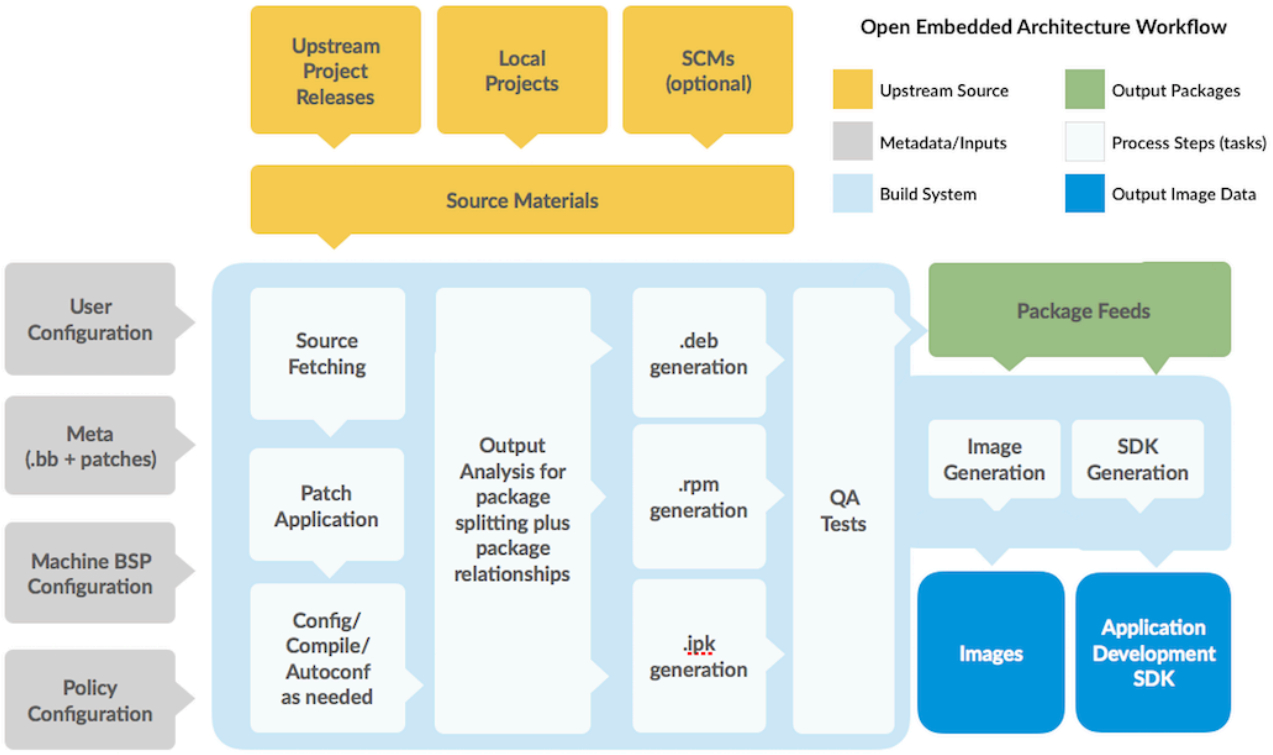


Figure 2.3: The General "Workflow" – How it All Works [25]

### 2.5.1 NXP Yocto

The NXP [26] i.MX release manifest repository contains the BSP (board support package) releases from NXP in form of an entire yocto build environment. This contains a set of multiple layers and base configuration available for building the BSP for supported hardware. This work is based and tested on the Hardknott and Kirkstone versions of the Linux release of the Yocto Project, with a custom layer on top to add support libraries for measurements.

## 2.6 ADASMark

The EEMBC® ADASMark benchmark suite is a performance measurement and optimization tool for automotive companies building next-generation advanced driver-assistance systems (ADAS) [27]. This project provides a benchmark implemented in OpenCL which stresses various forms of compute resources and allows to determine the optimal utilization of available compute resources.

This benchmark was thus chosen to test discovered optimal parameters in a more complex application which better models an actual, practical application. The ADASMark default pipeline can be seen in Fig. 2.4. It covers almost all common operations, including blurring, contour detection and even utilized a neural network to detect road signage.

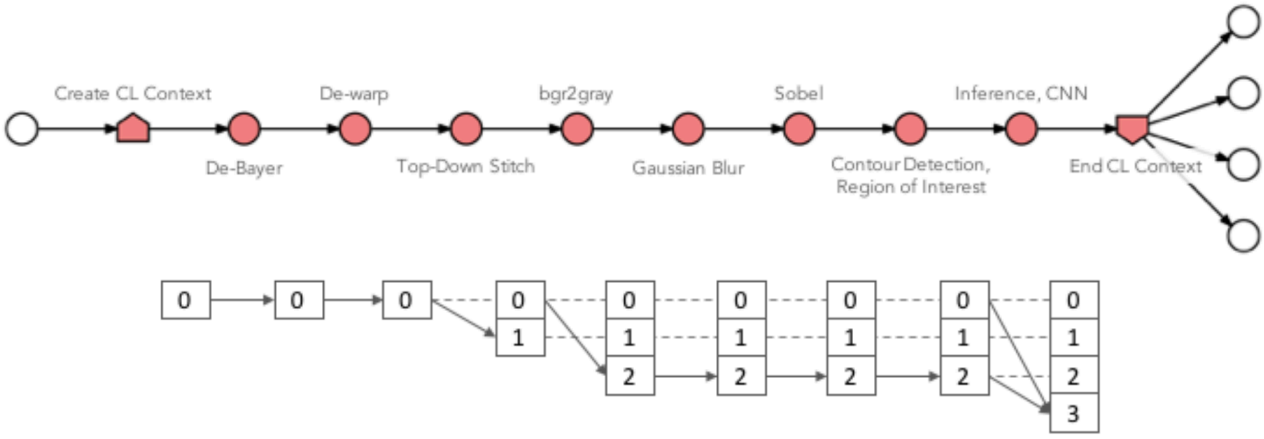


Figure 2.4: The ADASMark default pipeline used in the bechmark.

## 2.7 Ghidra

Ghidra [28] is a software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission. It can be used to help with understanding the performance characteristics of binary-only distributed software, and discover undocumented parameters which might help optimizing performance for specific hardware. A screenshot of the interface can be seen in Fig. 2.5.

## 2.8 Information resources

There are multiple kinds of information resources to study to further understand the system and all its parts. Here we list the major categories:

- Documentation – studying available documentation is an important step to discovering edge cases, limitations and unknown capabilities of the system. NXP provides a set of guide documents for the Yocto project, Linux usage and Graphics programming (including OpenCL). Often this documentation is incomplete, but still provides a lot of information about the system.
- Source code – OpenCL and Vulkan also have excellent documentation and pieces of source code which should be consulted to details about specific operations used in implementation. When documentation is lacking source code can fill in the details.

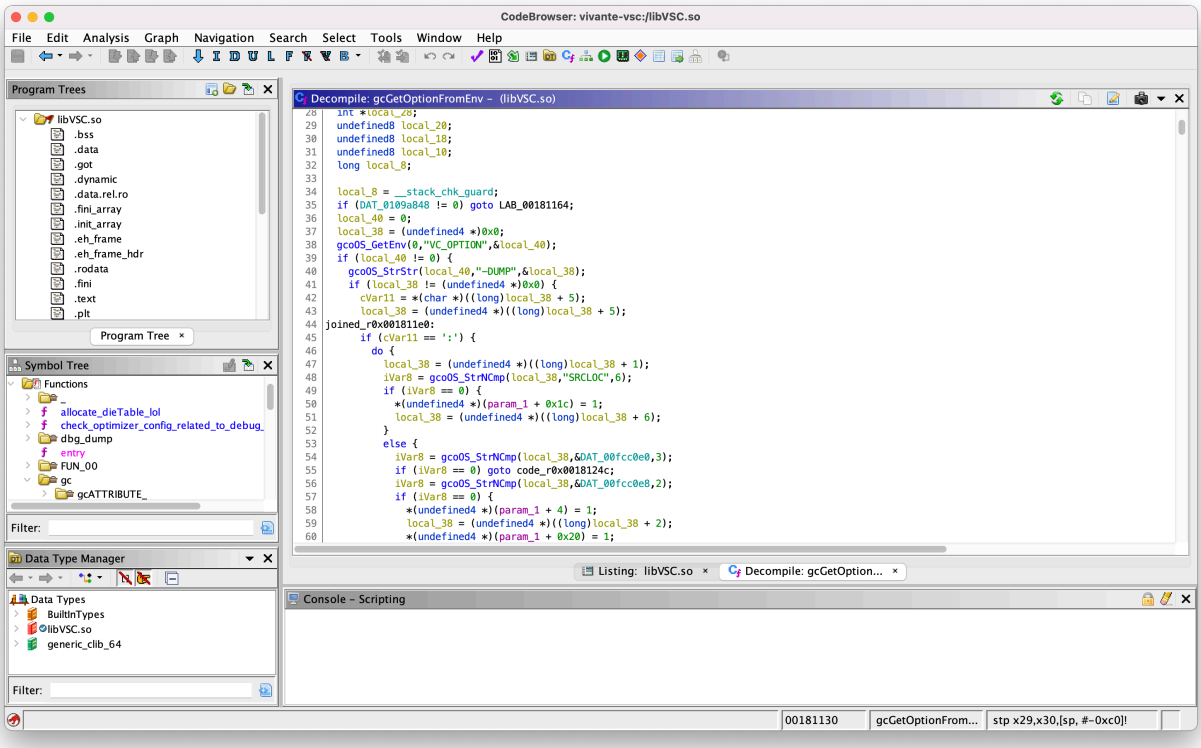


Figure 2.5: Screenshot of Ghidra interface.

- Reverse-engineering – We will also attempt to reverse-engineer proprietary drivers to understand their performance behavior and discover additional configuration flags or capabilities. Incomplete documentation paired with closed-source code requires this approach to understand the system.

# 3 Solution design

Multiple benchmarks were designed to test specific properties of the testbed with respect to energy efficiency and heat output. This chapter describes the high level design of the approach to measuring the effect of configuration parameters on heat output, including processes for booting and system management. The overview of the entire solution can be seen in Fig. 3.1.

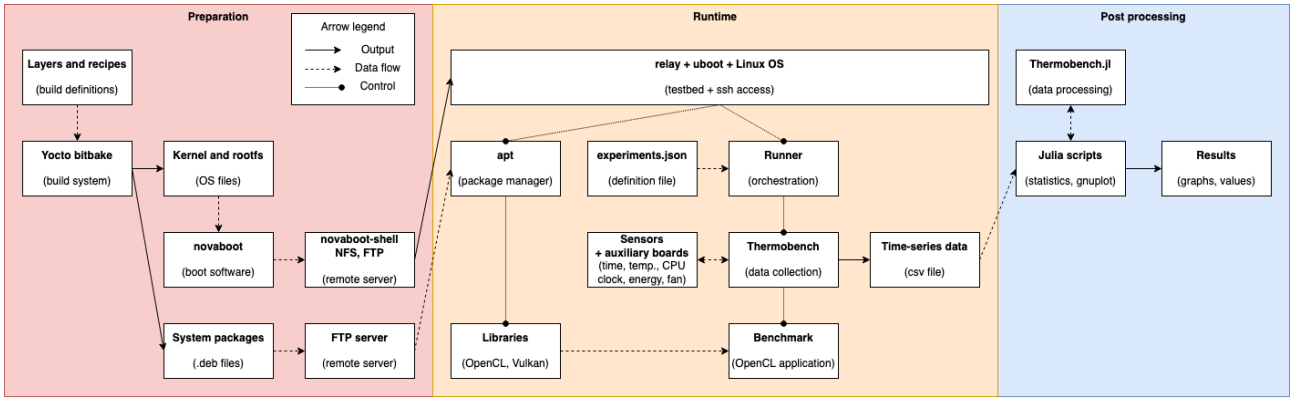


Figure 3.1: Solution design overview

## 3.1 Operating system distribution

We want a reproducible way to boot the same system repeatedly, for example after system crashes or when sharing the physical board with other researchers. Also with a convenient way to add new software by compiling it. We need an effective process for creating such system in a composable way.

The core of the Linux operating system is its kernel which is updated regularly, and was updated at least once during the creation of this work. It might be worthwhile to test the difference between consecutive versions.

It is also important to be able to test different drivers and libraries available within the ecosystem. An interesting comparison is open-source drivers vs. proprietary drivers, ICD loaders and the overall integrational stability of the system.

For these purposes the Yocto project was chosen, as it is already provided by NXP as an option. It allows to describe “recipes” to create a custom Linux distribution, especially by adding new ones or modifying existing ones. We will add software to the distribution provided by NXP by creating our own recipes.

## 3.2 Measurement automation and support libraries

We need reproducible benchmarks so that we can both iterate on their design and make sure our results are reproducible in the future. We also need to be able to cleanly define benchmarks and parameters.

We design a set of scripts to easily configure concrete values for configuration variables in a central definition file (experiments.json). These scripts apply values at appropriate places in

the operating system or environment variable. We also design a base abstraction over OpenCL and Vulkan to ease with implementation of benchmarks by providing a common library. This is compiled as part of the benchmark as a static library. It allows us to have a common code structure and command line parameters in each benchmark and focus primarily on the parts that differ in each benchmark.

## 3.3 Benchmarks

To study the properties of the platform we need to cover categories of floating-point and integer compute, memmory bandwidth and pure overhead. We design the following benchmarks in OpenCL and some benchmarks also in Vulkan:

- Mandelbrot – We use previously available manderbrot kernel program, which computes a 2D image of the mandelbrot fractal and expose as many parameters as possible to test their effect on the solution. We also implement this benchmark in both OpenCL and Vulkan.
- Clpeak integer compute – We utilize the integer compute benchmark from the clpeak suite to cover integer compute power.
- Clpeak memory bandwidth – We also utilize the memory bandwidth benchmark from the clpeak suite to measure raw memory throughput based on configuration and access patterns.
- Overhead – We design a benchmark which dispatches the simplest kernel possible without allowing the compiler to completely optimize it away. We use this benchmark to measure the overhead of dispatching work using the chosen API, so we implement it in both OpenCL and Vulkan.

We implement these benchmark in the Rust programming language [22] for its memory safety features and interoperability with the C language in which OpenCL and Vulkan APIs are officially available.

## 3.4 CPU/GPU work split

We compare the effectivity of GPU and CPU in specific workloads. We also attempt to perform a combined CPU and GPU execution where one part of the workload is performed on one processing unit, the other part on another processing unit, then combined into a final result.

We attempt to utilize PoCL to offload part of the workload to the CPU making use of the idle CPU time and then combining the partial results into a final result. We do not attempt to optimize parameters for each processing unit individually, instead we optimize for the most powerful chip (the GPU) and use the same parameters for the offload chips.

## 3.5 Application benchmark

A more comprehensive benchmark should be used to evaluate the solution. For this we choose ADASMark automotive benchmark, which includes different kinds of compute operations and overall fits the theme of thermal contrained environment.

We use the configuration capabilities of ADASMark and attempt to utilize vendor-specific implementations which happen to be available for our board.

# 4 Analysis and implementation

This chapter describes more focused analysis of available libraries and configuration parameters and details of the software support created to reliably perform benchmarks.

## 4.1 OS image preparation

We want to ensure reproducibility of the entire system, including the OS kernel and libraries, as well as to add additional packages which are not yet directly included. In this section we describe how the OS image and libraries are built and distributed onto the board.

The way to do this in the Yocto project is to add our own layer. The Yocto layer recipes produce one or more packages, which are later packaged into the `.deb` format and distributed onto the board through internal network through the `apt` package manager, as would be in a standard Linux distribution.

### 4.1.1 Yocto base layer

The very base layer provided by the Yocto project contains the base Linux kernel. The NXP repository is regularly updated as underlying layers update. We build the versions codenamed Hardknott and later Kirkstone. Initially the Hardknott version exhibited kernel crashes when running longer OpenCL tasks. Asking on the NXP forum [29] showed that the bug was already fixed in a newer version, Kirkstone. This new version did not exhibit crashes anymore. There was not a noticeable difference between these builts in measurement results (when they did not crash) but as Kirkstone includes a newer kernel and did not crash all results were collected on the Kirkstone build.

### 4.1.2 Edited Yocto recipes

To allow the system to work with libraries we want to add we need to edit some recipes. Specifically we require support for OpenCL ICD loader. These recipes, along with all source code, are available in the datadisk and online repository [1] at `system/meta-thermac`.

The `imx-gpu-viv` recipe provided by NXP had to be tweaked to correctly integrate with open-source libraries of the base system. Originally, this package claims to provide an OpenCL ICD loader, however, this isn't the case and the libraries are instead the direct Vivante OpenCL driver implementations.

To fix this, the custom layer mutates the recipe, removes the claim to be an ICD loader and requests the one provided directly by Khronos. Additionally, the OpenCL headers provided by this package are also removed, as they are already present in the dependecy chain of the ICD loader.

Finally, the library files needed to be renamed to vendor-specific variants so that the ICD loader would not clash with them. This was done by appending the recipe with a script which performs this move and patches the ELF soname attribute to change the library name so that it is correctly recognized and handled by the Linux library ecosystem.

The `imx-dpu-g2d` recipe also needed mutation to add dependency on ICD loader, as it was previously reliant on the `imx-gpu-viv` recipe which was no longer providing it.

We wanted to compare proprietary kernel graphics drivers against current open-source ones. However, while the required steps to build the proprietary drivers are still present in the `kernel-module-imx-gpu-viv` the produced Linux distribution is not supported anymore and the kernel module crashes when work is issued. Therefore, we used only the open-source drivers.

### 4.1.3 Added Yocto recipes

To allow using the CPU as an OpenCL target platform we need a driver capable of targeting a CPU. For this purpose the PoCL library was chosen. The first requirement was to support ICD libraries which was achieved by changing the `imx-gpu-viv` recipe as described above. Recipes are available in the datadisk and repository.

The PoCL recipe did not exist in the open-source recipe repository so it was created. It includes a source-code patch to solve a type-width inconsistency, specifically around signed pointer type width in the C language versus in the OpenCL shader program C dialect. This recipe instructs the build system to optimize PoCL for the specific target architecture, build in release mode and include support for ICD loaders.

Next, we need the final application benchmark, which requires additional libraries. Since we chose ADASMark all required libraries were already present in the Yocto stack. The recipe for ADASMark was split into two parts – the ADASMark benchmark, which is distributed as source code to be built. And the AuZone pre-trained neural classification net, which is distributed as part of ADASMark benchmark as proprietary prebuilt binary files.

The main recipe futher generates two packages – `adasmark` and `adasmark-data`. The second package contains the official data distributed with the benchmark, such as the input video and OpenCL kernels. The main package contains libraries and binaries built for the target system, including a source-code patch to fix errata 1 as mentioned in the Adasmak User Guide attached with the benchmark.

Finally, the benchmark also provides vendor-optimized versions of kernels and nodes, especially for NXP i.MX8 board, which we compiled to be compared against the base implementation.

### 4.1.4 Booting the image on the board

To provide convenience and resilience the board is booted from a server within the network and connection to the board happens remotely over the SSH protocol, using the server as an intermediate hop. The system is booted using U-boot and facilitated using novaboot and novaboot-shell.

The root filesystem is served from userspace NFS server [12] because the physical server is shared with other researchers. This userspace NFS project had to be additonally patched to ignore chown and chmod requests from the client as in userspace the server is unable to change server-side ownership when not running as the root user. This was in particular needed to allow dpkg to function at all. The patch is available in the datadisk and online repository [1] at `system/misc/unfs3-chown.patch`.

Most software worked correctly with this workaround, however some software included sanity checks which first ran chown/chmod operation and immediately checked whether the new owner/permissions were what was expected. For this software no additional workarounds were made.

The following block shows an example of the boot log output. Lines containing `--snip--` represent one or more skipped lines:

```
$ lavusedu@ritchie:~/CIIRC/yocto-output$ ./boot.sh third-root
novaboot: Effective options: third-root.nova -ssh=imx8@rtime
novaboot: Running: ssh 'imx8@rtime' get-config
novaboot: Received configuration from the server:
--snip--
novaboot: Running: ssh -tt -M -S /run/user/474497/novaboot2806368 'imx8@rtime' consol
novaboot-shell: Connected
--snip--
novaboot: Resetting the test box...
--snip--
novaboot: Waiting for U-Boot prompt...
--snip--
novaboot: Serial line interaction (press Ctrl-C to interrupt)...
--snip--

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[   0.000000] Linux version 5.15.32-lts-next+gfa6c3168595c (oe-user@oe-host)
(aarch64-poky-linux-gcc (GCC) 11.2.0, GNU ld (GNU Binutils) 2.38.20220313)
#1 SMP PREEMPT Tue Jun 7 02:34:46 UTC 2022
[    0.000000] Machine model: Freescale i.MX8QM MEK
--snip--
[    9.191732] Run /sbin/init as init process
[   10.343008] systemd[1]: systemd 250.4-1-gc3aead5+ running in system mode
--snip--
NXP i.MX Release Distro 5.15-kirkstone imx8qmmek ttyLP0

imx8qmmek login:
```

## 4.2 Automation

There are many libraries and other support software between the benchmarks and the hardware. Serving to abstract away hardware details, provide uniformization and make development easier. This software is often configurable or tweakable to allow the user to optimize for their target system. The following subsections describe how abstraction was built to automatize runtime configuration and benchmark development.

### 4.2.1 Testbed

The Turbot board (blue board visible in Fig. 2.1) controls the testbed, including the fan mounted on the MEK board. It collects data such as ambient temperature and energy consumption. The MEK board can access these controls and data over SSH during measurements.

### 4.2.2 Thermobench

Thermobench is the main way to collect results of measurements. It launches a given benchmark process, collects events from sensors and from the output of the benchmark itself and stores all information in a CSV file. This includes:

- time since beginning of the measurement
- CPU load
- CPU temperature
- CPU frequency
- GPU frequency
- energy consumed by the board (through Turbot board)
- ambient temperature (through Turbot board)
- inter-workload markers which the benchmark benchmark reports, such as when individual kernels are dispatched and when they finish.

All these quantities are timestamped and preserved so that it is possible to derive meaningful information from the entire runtime of the benchmark. The data from the CSV file can be visualized as can be seen later in Fig. 4.1.

### 4.2.3 Measurement runner

The measurement runner is a Python script which facilitates configuring and launching thermobench and other runners and benchrmaks underneath them. This script is available in the datadisk and online repository [1] at `experiments/runner/runner.py`.

It implements all Linux configuration possiblities considered in this work, as well as as passing environment and CLI variables to the subprocesses. In addition, it allows creating permutations of arguments to measure how individual arguments interact with each other within an benchmark.

**JSON description** To fully separate the implementation from measurement description and to allow clear and conscise understanding of what each measurement does, a custom JSON description file is used. This file is available next to the runner at `experiments/runner/experiments.json`.

It allows defining CLI commands to execute, specify variable substitution, environment parameters and permutations of variables. The runner reads this configuration and executes a measurement as defined.

### 4.2.4 Result analyzer

The result analyzer is a set of Julia scripts, making use of the Thermobench Julia library, to analyze Thermobench CSV output files and visualize them using Gnuplot. These scripts are available in the datadisk and online repository [1] at `results/thermobench/{common,entry,plotting}.j`

All graphs in this work were drawn using these scripts and should be reproducible using only the corresponding CSV files, which are persisted in git repository. An example of such a graph can be seen in Fig. 4.1.

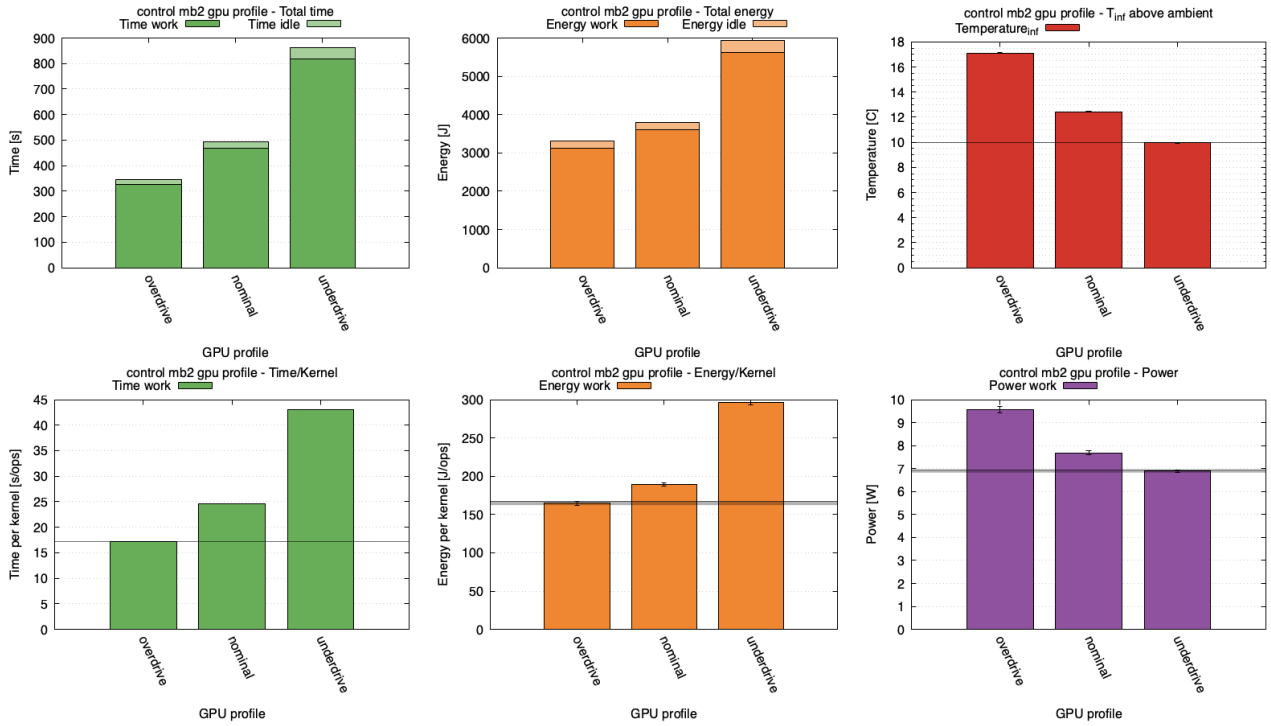


Figure 4.1: An example of the kind of graph that is output from the result analyzer.

### 4.2.5 Common libraries

To ease development of benchmark code a common library was created on top of Rust OpenCL FFI bindings. This library is split into two Rust crates. These libraries are available in the datadisk and online repository [1] at `experiments/{opencl-ffi, opencl-lib}`.

The Vulkayes Vulkan library was also used to implement Vulkan versions of benchmarks.

**OpenCL FFI bindings** The Rust bindgen crate [30] was used to generate bindings to the OpenCL libraries based on OpenCL headers [31]. From these headers specific programming symbols were selected and are generated into Rust interface definitions.

The bindings use Rust feature gates to allow selecting which OpenCL version the code targets. This depends on the target system libraries or the ICD loader version, so it needs to be selected for each project individually.

**OpenCL Rust wrapper** The ecosystem was missing a high level Rust wrapper for OpenCL API safely exposing desired APIs. Most available crates are lower level than was desired for this work.

The OpenCL API, while platform agnostic, is tied to its canonical representation in the C programming language. While the API contract can define rules and requirements for interacting with the API, the C language itself cannot expres most of these requirements explicity and they are only included in the specification for the developers to find and fulfill.

Rust provides many more opportunities to encode API requirements and contracts into the programmatic source code itself and together with other benefits it greatly reduces the strain on the programmer to uphold the requirements.

The abstraction library creates such a layer on top of the FFI bindings and avoids undefined behavior at little-to-none runtime cost. Among other quality-of-life benefits, the library provides error definitions with proper documentation encoded in Rust tagged enums, inherent

null pointer safety thanks to Rust references, destructors for automatic memory management and proper type safety in wrapper functions.

Overall, this library makes the benchmark codes more readable by reducing the boilerplate needed to initialize and cleanup objects, as well as making sure the calls to the API are memory safe.

## 4.3 System configuration

This section reviews configuration available in the testbed, operating system and libraries. Configuration of specific benchmarks is described in a later section.

The results of benchmarks are affect by many factors. Some which we can change and see the effect on the results. See Tbl. 4.1 for an overview.

Table 4.1: System configuration parameters overview

| Parameter | Values | Description |
|---|---|---|
| Fan speed | 0%-100% | Fixed at **50%**. |
| Process niceness | -20 - 19 | Fixed at **-20**. |
| Process scheduler | default, idle, batch, fifo, round-robin | Fixed at **batch**. |
| CPU profile (cores) | 4x A53, 2x A72 | Default all. |
| GPU profile (DVFS) | overdrive, nominal, underdrive | Default overdrive. |
| VIV_MGPU_AFFINITY | 0, 1:0, 1:1 | Default 0 – available chips split the work. In 1:{0,1} mode the second number decides which chip of multi-chip GPU runs the work. |
| POCL_AFFINITY | 0, 1 | Fixed at **1**. |

### 4.3.1 Testbed

The testbed mainly provides access to sensors, but one interesting configurable part is the fan attached to the SoC of the board. This fan has been fixed at 50% to provide reasonable cooling while keeping the noise low.

### 4.3.2 Operating system

There are multiple things which the operating system can control. All of these can be readily studied from the Linux documentation and its source code.

**Process niceness** Set to -20 i.e. the highest priority. Since the benchmark is the only CPU-intensive process running at a time this doesn't have a noticeable effect on the performance, but it is a good practice to set it nontheless.

**Process scheduler** After evaluating different scheduler configurations and their effect on performance and variance this parameter was fixed at `SCHED_BATCH`.

Realtime schedulers produced much higher variance in measurements and were ruled out right away. The CFS in its default configuration performend slightly worse than its batch configuration, however it was still much better than realtime schedulers.

**CPU core hotplugging** By default, all cores are enabled. It is possible to turn off individual CPU cores.

**GPU dynamic voltage and frequency scaling** The GPU boots in `overdrive` mode. One of our benchmarks comfirmed that DVFS is more efficient than software throttling, as long as lower performance is a desirable tradeoff.

### 4.3.3 Reverse-engineering libVSC

libVSC is the underlying library for all NXP userspace drivers. It contains many utility and driver-like functions for manipulating the Vivante hardware, as well as a compiler to produce GPU shader machine code from OpenCL kernels and SPIR-V bytecode. This library is distribuded as a closed-source binary blob. Some features are documented in the available i.MX8 graphics guide document, others can only be understood by reverse-engineering the drivers.

This library is not open-source and has no open documentation. The only available information is about high-level APIs like OpenCL and Vulkan, but this library is not mentioned. However, this library provides most of the shared functionality of the Vivante GPU userspace drivers and as such is the central piece of the drivers.

Interestingly, this library also contains code (in function called `gcoOS_DetectProcessByName`) which detects the name of the running process and attempts to match it against a known set of binary names. When stored in the binary, these strings use a rudimentary encryption in the form of flipping all bits of the string byte sequence, presumably to avoid suspicion when dumping library strings. The bits are flipped back when the comparison with the actual binary name is made, and if a match occurs it is stored in the global state of the application. It is unknown to us how this information is later utilized.

Furthermore, this library reads and reacts to multiple different environment variables, which can be used to change behavior of the compiler or to enable debugging output (`VIV_DEBUG`). Unfortunately most of these are undocumented and not very widely supported inside the library as sometimes they cause the running binary to segfault instead of presenting debugging output. The compiler is already pre-set to compile to the highest optimization level.

Next, there are documented variables `VIV_PROFILE` and `VP_OUTPUT` which can be used to output tracing information to be later consumed by Vivante tools to visualize perfomance and bottlenecks. The format of these files is also proprietary.

On devices with a GPU which is comprised of multiple internally-independent chips, such as our i.MX8QM SoC, it is possible to control the affinity of GPU using the `VIV_MGPU_AFFINITY` variable, as documented by the i.MX Graphics User's Guide. This has a noticeable impact on the performance and temperature characteristics of the SoC. One very interesting finding is that the thermal characteristics of the separate units the GPU is composed of is not the same. This can be seen in Fig. 5.12

Finally, Vulkan exposes even more control to the application and thus supports even less environment variables. However, one variable which remains is the `VIV_MGPU_AFFINITY` variable for controling GPU affinity. Unfortunately, the NXP driver implementation does not currently process this variable and thus Vulkan always runs only on one of the multi-GPU unit. This is not possible to be configured through environment variables, nor through Vulkan extensions available in the drivers. Even with support from NXP TechSupport it was not resolved and so Vulkan performance can only be measured on one chip [32].

### 4.3.4 PoCL library configuration

PoCL library respects multitude of environment variables, mostly for debugging. Most important variables control the pinning of each execution thread in its pool to a specific core (`POCL_AFFINITY`) and number of threads in its pool (`POCL_MAX_PTHREAD_COUNT`). The former controls CPU affinity, which has effect on lowering the variance of execution time as the CPU core composition of this board is heterogeneous (LITTLE and big cores combined) and the Linux kernel migrates threads a lot, causing noticeable execution time variance. Setting thread affinity, i.e. pinning, ensures that even when offloading work to CPU the execution time variace remains low.

Documentation and source code is available online [33].

## 4.4 Benchmarks

The main programming language for implementation of benchmarks is Rust [22]. It has FFI (foreign function interface) bindings to the OpenCL and Vulkan libraries and provides advanced memory safety guarantees to the programmer. Each benchmark includes tweakable parameters to test their effects on the performance and energy characteristics of the benchmark. Common parameters are handled by a shared library which handles generalized input parsing, work dispatching and instrumentation.

This section describes implementation of individual benchmarks.

### 4.4.1 Mandelbrot

The mandelbrot benchmark is based on computing 2D black and white visualization of the mandelbrot fractal – see Fig. 4.2.

This is a compute-bound benchmark performing floating-point computations. The code for this benchmark is available in the datadisk and online repository [1] at `experiments/experiments/src/bin/mb2.rs`. For the benchmark to give meaningful results, we had to solve the following problems:

- Prevent the kernel compiler from optimizing the computation away (while still allowing optimizations to happen).
- Make every pixels run the same amount of iterations to make the work uniform.

This benchmark is especially well suited to represent common image computation tasks as it incorporates 2D domain, floating-point calculations and independent work which lends itself to parallelization both inside and outside of the processing unit. Tbl. 4.2 shows an overview of configurable parameters of this benchmark.

Table 4.2: Mandelbrot benchmark specific parameters

| Parameter | Values | Description |
|---|---|---|
| Problem size | width * height | Size of the computed image. Fixed at **3072x3072**. |
| Work group size | x * y | Size of one work group. The area is required to be at most 1024. By default **32x1** is used. For CPU/GPU work split each dimension should be a divisor of corresponding problem size dimension. |
| Maximum pixel iterations | *number* | Maximum number of iterations to computer per pixel. Fixed at **8192**. |
| Escape radius | *number* | Radius after which a pixel computation escaped and the mandelbrot is computed. Fixed at **infinity** to force all pixels to perform the same amount of work. |

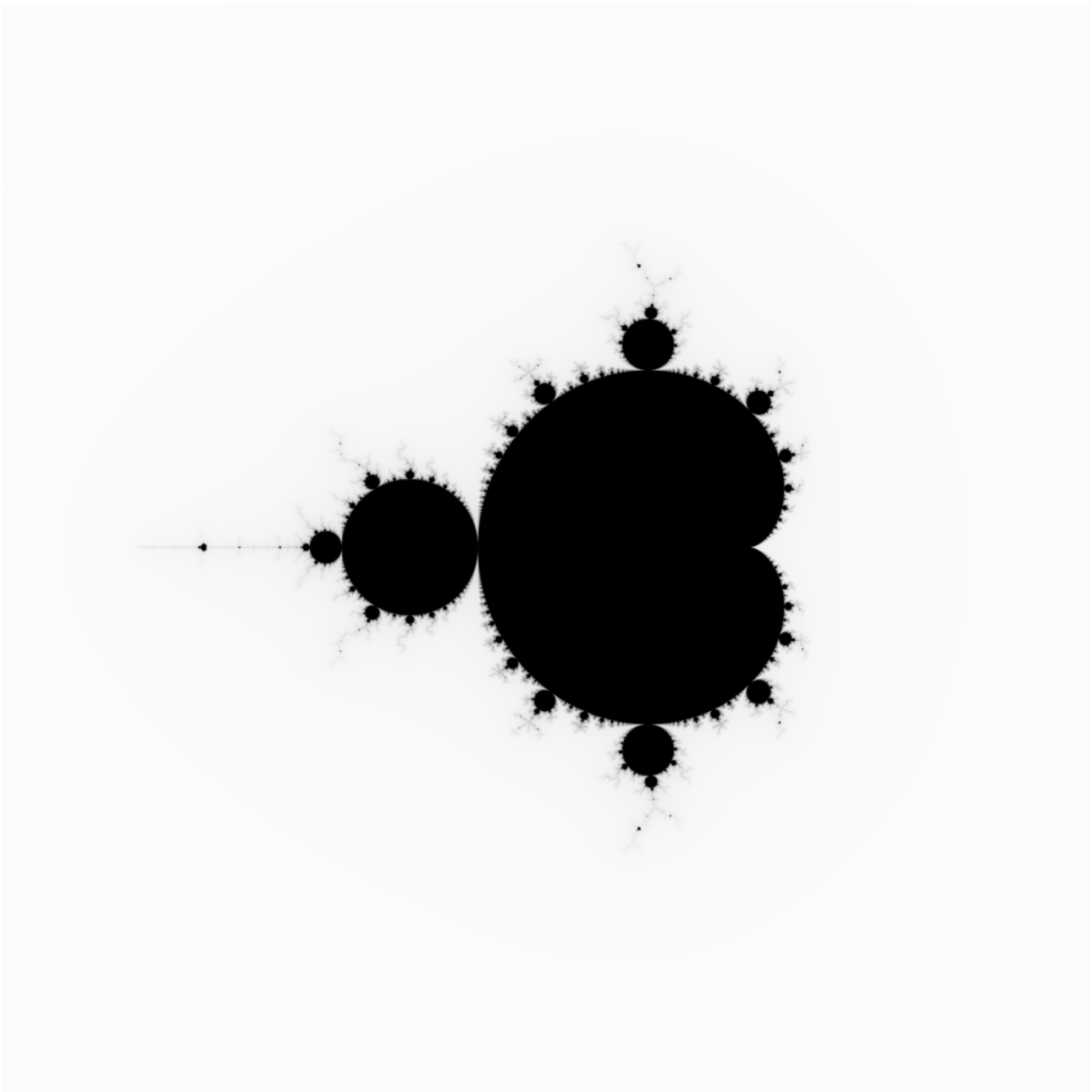

Figure 4.2: The computed 2D mandelbrot fractal visualized.

### 4.4.2 Clpeak memory bandwidth

The clpeak memory bandwidth benchmark is ported from the clpeak test suite [34] with small modification to the way results are stored in the output buffer. The code for this benchmark is available in the datadisk and online repository [1] at `experiments/experiments/src/bin/clpeak_global_bandwidth.rs`. Originally, this benchmark serves as a way to measure memory performance. It is implemented as reading values from input memory, running very cheap operations on these values and storing the results into output memory. These values can be read individually or as vectors (multiple memory-consecutive values at once). Tbl. 4.3 shows an overview of configurable parameters of this benchmark. The memory read pattern is visualized in Fig. 4.3.

Table 4.3: Clpeak memory bandwidth benchmark specific parameters

| Parameter | Values | Description |
|---|---|---|
| Problem size | *number* | Size of the vector. Fixed at **16777216**. |
| Work group size | **32**, 64, 128, 256, 512, 1024 | Size of one work group. Smaller value cause huge performance loss due to dispatch overhead. |
| Vector representation | **v1**, v2, v4, v8, 16 | Vector element representation in the shaders. |
| Offset type | **local**, global | The pattern in which one work item accesses memory. |

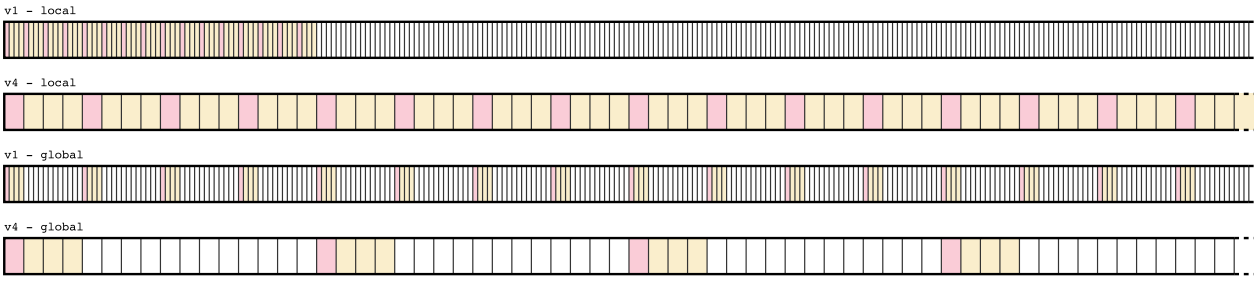


Figure 4.3: Memory read pattern for the Clpeak memory bandwidth benchmark. Rows visualize memory of the input buffer. The input buffer is of the same size with every vector representation. Red rectangles correspond to the (vector sized) elements accessed by the first work item (and consequently stored in the first element of the output buffer). The yellow rectangles correspond to the elements accessed by the work items of the first work group (the first group includes the red rectangle as well). White rectangles are accessed by remaining work items from remaining work groups.

### 4.4.3 Clpeak integer compute

This benchmark is inspired by the `compute_integer` available in the clpeak test suite [34] with major modifications – for each element of the output array it runs the same amount of work. The code for this benchmark is available in the datadisk and online repository [1] at `experiments/experiments/src/bin/int_compute.rs`.

This benchmark works on 32bit signed integers and performs multiply-add (MAD) operation 1024 times per output integer. This is a better simulation of real-work application than the original benchmark, which runs the same amount of MAD operations per work-item, because one work-item can represent multiple individual integers when bigger vector representation is chosen.

It represents operations which are compute intensive in the integer domain for comparison with floating-point computations. Tbl. 4.4 shows an overview of configurable parameters of this benchmark.

Table 4.4: Clpeak integer compute benchmark specific parameters

| Parameter | Values | Description |
|---|---|---|
| Problem size | *number* | Size of the vector. Fixed at **16777216**. |
| Work group size | **32**, 64, 128, 256, 512, 1024 | Size of one work group. Smaller value cause huge performance loss due to dispatch overhead. |
| Vector representation | **v1**, v2, v4, v8, 16 | Vector element representation in the shaders. The greater this representation, the more work one work-item does, and the less work items there are in total. |

### 4.4.4 Overhead

The code for this benchmark is available in the datadisk and online repository [1] at `experiments/experiments/src/bin/overhead.rs`.

This benchmark executes a kernel with the following code:

```
__kernel void overhead(__global char* out) {
    volatile int a = get_global_id(0);
    return;
}
```

This code is very simple and has almost no performance cost, however due to the use of `volatile` the GPU is required to run it in case of any side effects which it potentially could have. Dispatching this kernel 50 000 times in one iteration allows us to measure the overhead of the dispatch.

### 4.4.5 Vulkan

The Mandelbrot and Overhead benchmarks are also implemented in Vulkan. This implementation is available in the same place as the OpenCL implementation. They use the same kernel code adapted to GLSL. One specific of the Vulkan API is that the local work group size can be specified in the kernel shader itself rather than in the dispatch command. However, kernel shaders are also required to support specialization constants. These were utilized to bridge the gap and allow the Vulkan benchmark to behave the same way as the OpenCL one – to specify the work group sizes on the CLI and pass them into the kernel shader programs at compile time.

However, due to the increased complexity of the driver with respect to OpenCL, lack of open-source alternatives and bugs in the proprietary drivers it requires more thorough testing to develop and deploy than OpenCL does. For example, Vulkan is currently not able to utilize both GPU cores available on the board. Attempts to solve this with the NXP support in their community forum resulted in them acknowleding the bug but it has not been fixed yet [32].

### 4.4.6 CPU/GPU work split

The CPU/GPU work split ratio controls the factor of the overall OpenCL work that is offloaded onto the CPU. For example, 0% means all OpenCL work runs on the GPU. Ratio of 20% means 80% (rounded up to nearest work-group size multiple) of the OpenCL work runs on the GPU in parallel to 20% (rounded down to nearest work-group size multiple) of the work being run on the CPU. The split always happens in one dimension. For one-dimensional benchmarks this is obvious, multi-dimensional benchmarks (such as the 2D mandelbrot) offer an additional parameter which controls whether the work is split on the x axis or the y axis.

The implementation is based on the installable client driver. Both the GPU and the CPU OpenCL userspace drivers are installed alongside each other and the loader is used to provide runtime ability to select which platform to use. The CPU and the GPU drivers are provided as two separate platforms. Instances of both platforms are created and initialized using the same kernels and settings. Then respective work portion is dispatched to each device using the OpencL `clEnqueueNDRangeKernel` function, specifying the work size and offset. The final buffer is of the same size for both parts, but each device only computes part of the output and the buffers are merged by the CPU at the end.

### 4.4.7 ADASMark

The ADASMark benchmark is a pipeline composed of separate nodes. Each node can be configured separately and that allows us to run certain nodes on the CPU using PoCL, including in work split mode, in the same manner as above in Sec. 4.4.6. Work split ratio is limited to 25% increments by ADASMark, so we choose the closes percentage matching CPU/GPU work split results.

Unfortunately, the vendor-specific patches specifically for the i.MX8 board which were distributed with the ADASMark codebase produced runtime errors and did not work. Therefore no comparison with the base implementation could be made.

Another bug found was that running a certain combination of nodes in work split mode caused the benchmarking probes to report invalid data, so at most one node at a time uses CPU/GPU work split.

# 5 Results

This chapter describes methodology used to run benchmarks described in the previous chapter and presents visualized results of running these benchmarks with varying parameters on our testbed.

## 5.1 Evaluation methodology

The results are presented in the form of graphs. The most common graph format used here can be seen in the example Fig. 4.1. On each subgraph the X axis is the parameter or set of parameters being tested. If there is a combination of parameters it is joined in by $\star$ symbol

The graphs in the first column (green bars) show time on the Y axis. The upper graph "Total time" shows total time of the measurement. The lower graph "Time/Kernel" shows time spent on one kernel by splitting full execution into individual kernel runs, removing first and last measurement and computing sample mean. Error bars show 95% confidence interval. The high error margin of the lowest data point is visualized as a horizontal line to allow easier visual comparision with other data points in the same graph.

The "Total energy" and "Energy/Kernel" graphs (middle colum, orange) show Energy on the Y axis instead of Time.

Upper-right (red) graph "T_inf above ambient" shows the T_inf as calculated by Thermobench. This is also known as steady-state temperature. It is calculated by fitting the measured temperature time series with an exponential function and taking the limit value for infinite time. Error bars show confidence itervals as reported by Thermobench `multi_fit` function.

Finally the lower-right (purple) Power graph displays power consumption over the benchmark runtime in the Y axis. Each individual Kernel run energy is divided by its run time and then sample mean is calculated. Error bars show 95% confidence interval.

## 5.2 Microbenchmarks

In this subsection, we list and comment the results of benchmarks described in Sec. 4.4 with the exception of ADASMark which comes in the following subsection.

### 5.2.1 CPU profile

Here we vary the CPU profile parameter and measure its effect. All of mandelbrot (Fig. 5.1), integer compute (Fig. 5.2) and memory bandwidth (Fig. 5.3) benchmarks achieve lowest temperature when all cores are enabled. Power consumption remains within the same confidence interval.

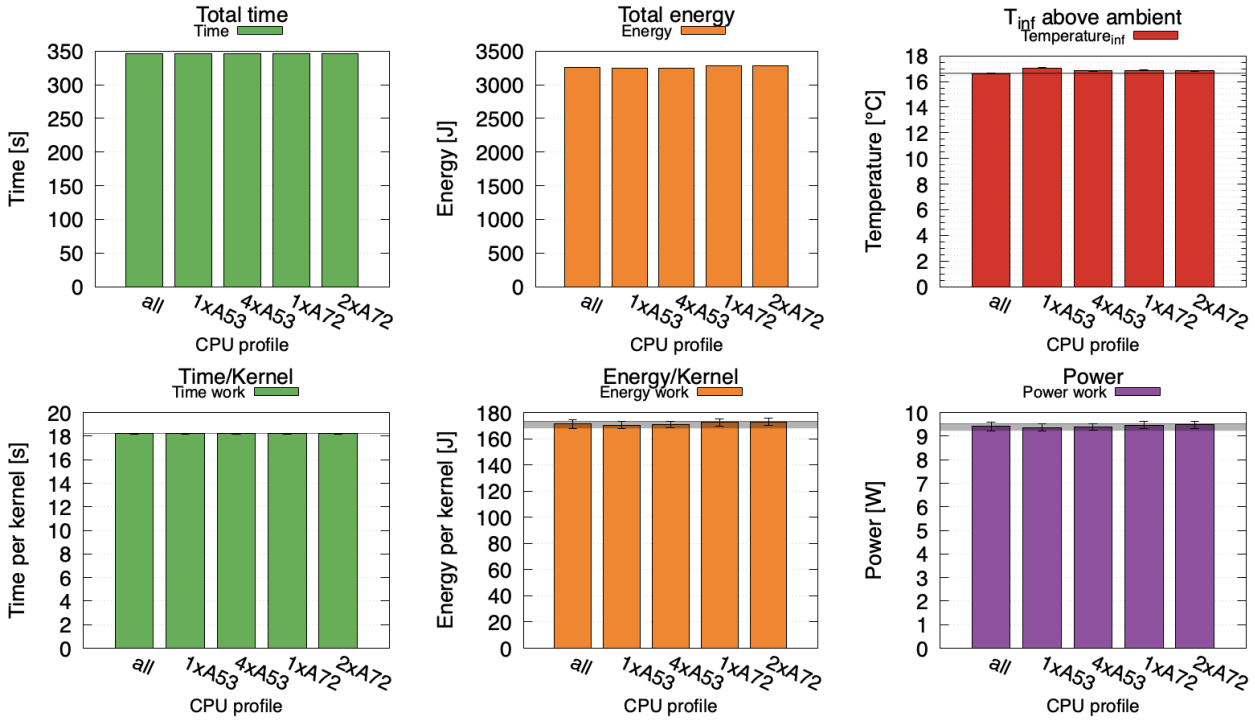


Figure 5.1: Varying the so called CPU profile does not have a noticeable impact on the performance or energy consumption of the mandelbrot GPU-only benchmark.

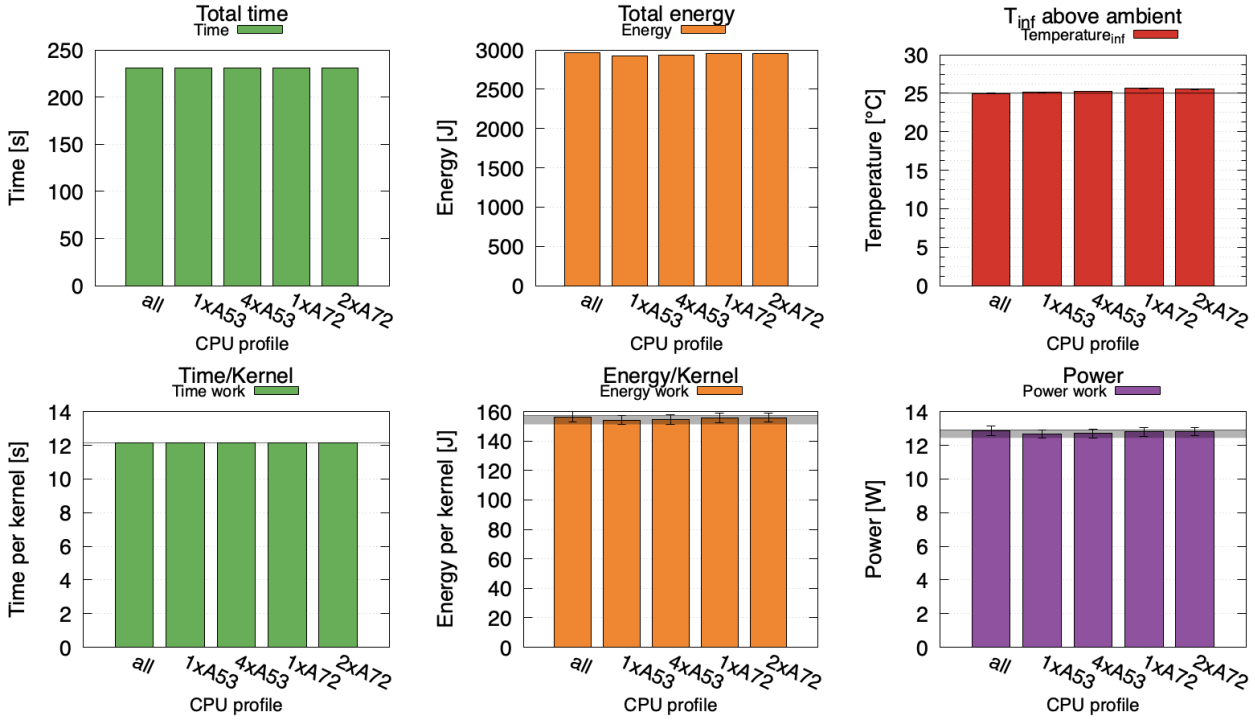


Figure 5.2: In integer compute we see little effect of CPU profile on measured quantities.

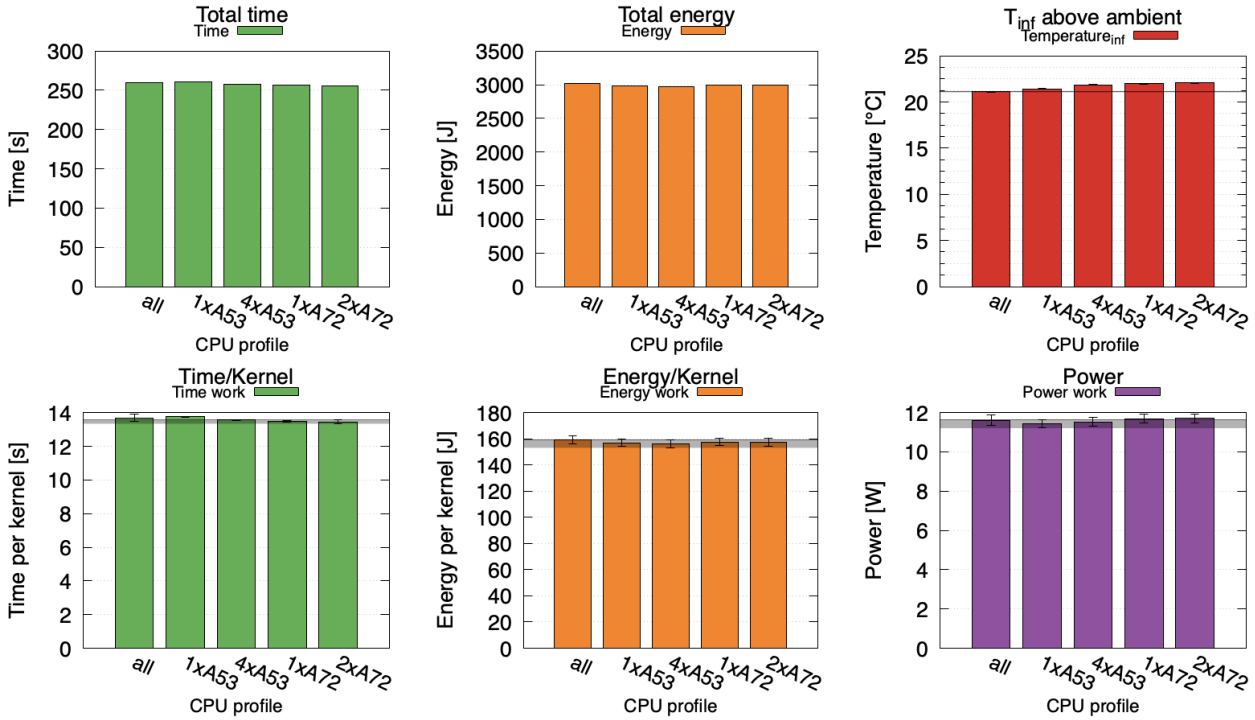


Figure 5.3: In memory bandwidth benchmark disabling CPU cores increases maximum temperature up to 5% but reports the same power draw.

### 5.2.2 GPU profile

In Fig. 5.4 we see expected behavior – lower GPU frequency increases run time but also decreases temperature and power. We can also see that the total consumed energy is higher for lower frequencies. This means that while lower frequencies are good for temperature reduction, they would be bad for battery powered devices.

In Fig. 5.5 we can see that the memory-bound nature of the benchmark results in energy consumption being the same for a small increase of execution time and a bigger decrease in temperature. This means that for memory-bound work it might be worth the small run time increase, especially if the data processing is bound by incoming data stream anyway, such as with live video feeds.

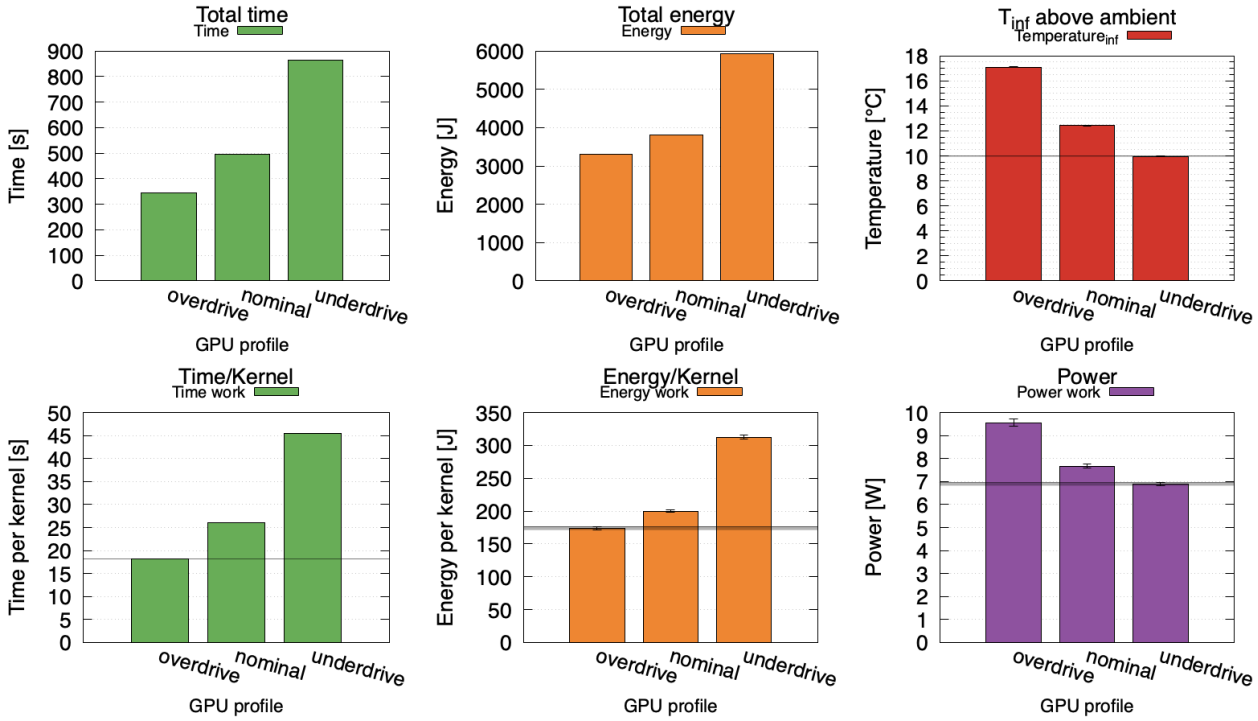


Figure 5.4: Mandelbrot executed with different GPU profiles shows no surprises.

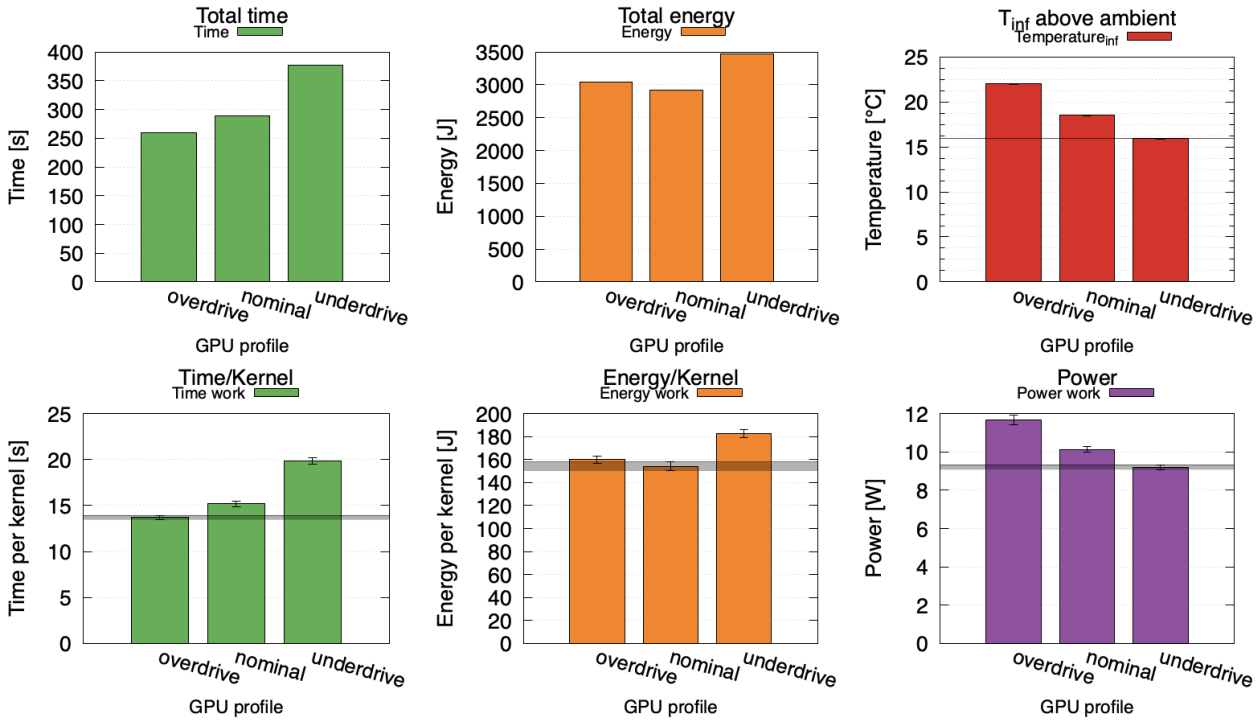


Figure 5.5: Clpeak memory benchmark benchmark – same energy consumption at `nominal` for 10% more time per kernel but almost 15% reduction in temperature.

### 5.2.3 Work group size

The work group size measurements (Fig. 5.6) show that most of the settings outside of the recommended vaue 32x1 have detrimental effect. Of note is that in 2D workload, setting the sizes so that the area is too big results in increase of maximum temperature.

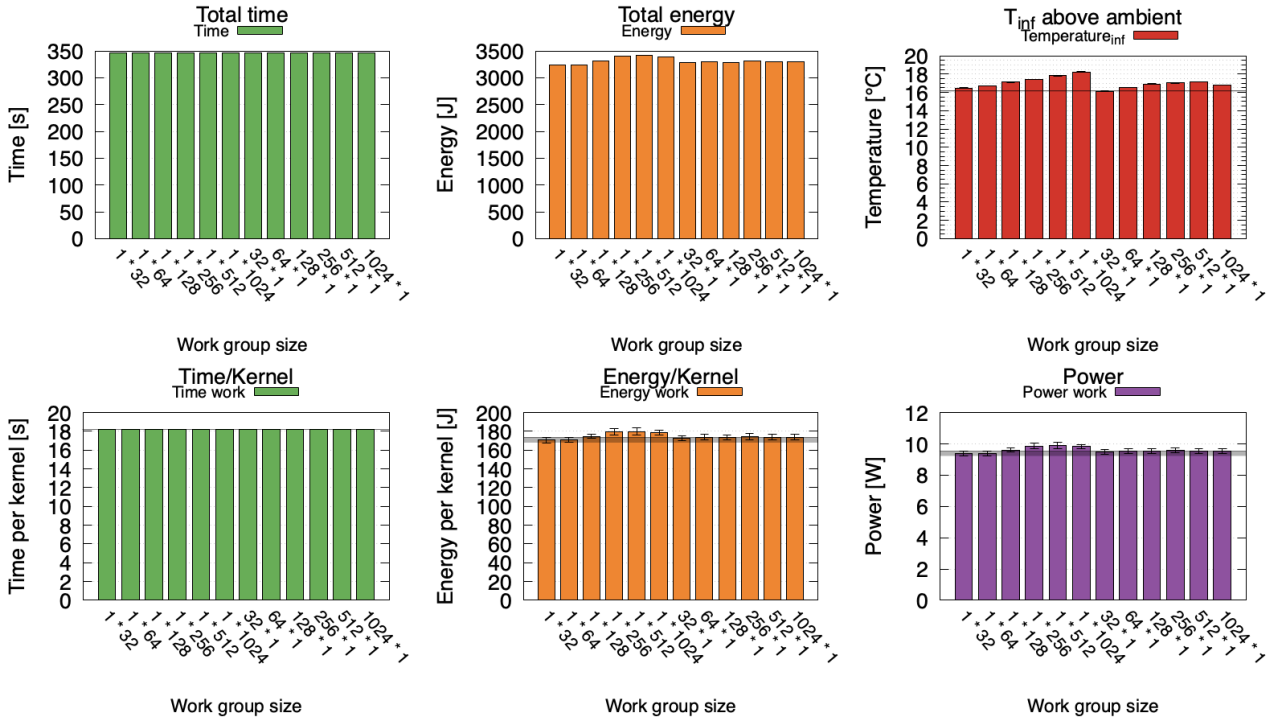


Figure 5.6: Effect of changing work group size for Mandelbrot benchmark.

In Fig. 5.7 we can see a grid plot of (top-left) time/kernel, (top-right) energy/kernel, (bottom-left) temperature and (bottom-right) power. We can see that 1x32 and 32x1 are two of the best choices. Thus by using the size recommended in the documentation we can achieve optimal results.

### 5.2.4 Memory access pattern

In the Clpeak integer compute benchmark (Fig. 5.8) increasing the vector size results in greater performance, slightly increased power consumption and higher temperature. As this benchmark is compute-bound increasing the vector size reduces dispatch overhead.

In the Clpeak memory bandwidth benchmark in Fig. 5.9 we see a different story. While increasing the vector size has a similar effect as above, at greater sizes it is counteracted by one work-item having to allocate more register space and having to access more memory, possibly causing cache pressure which reduces performance.

This benchmark also contains two different grouping patterns, local and global (see Fig. 5.10). Expectedly it is clear that local grouping is much faster, likely thanks to cache, as the access pattern is more cache friendly.

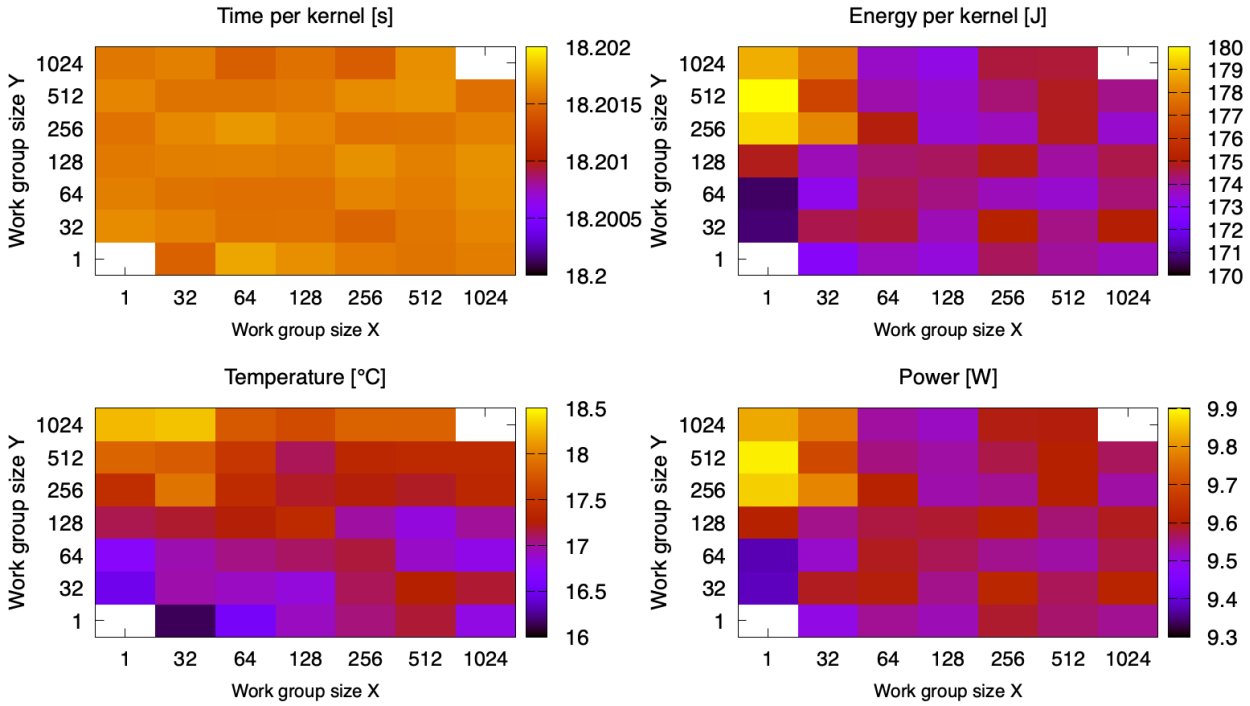


Figure 5.7: Grid plot shows that 32x1 has the best temperature behavior with time almost identical among all variants on mandelbrot.

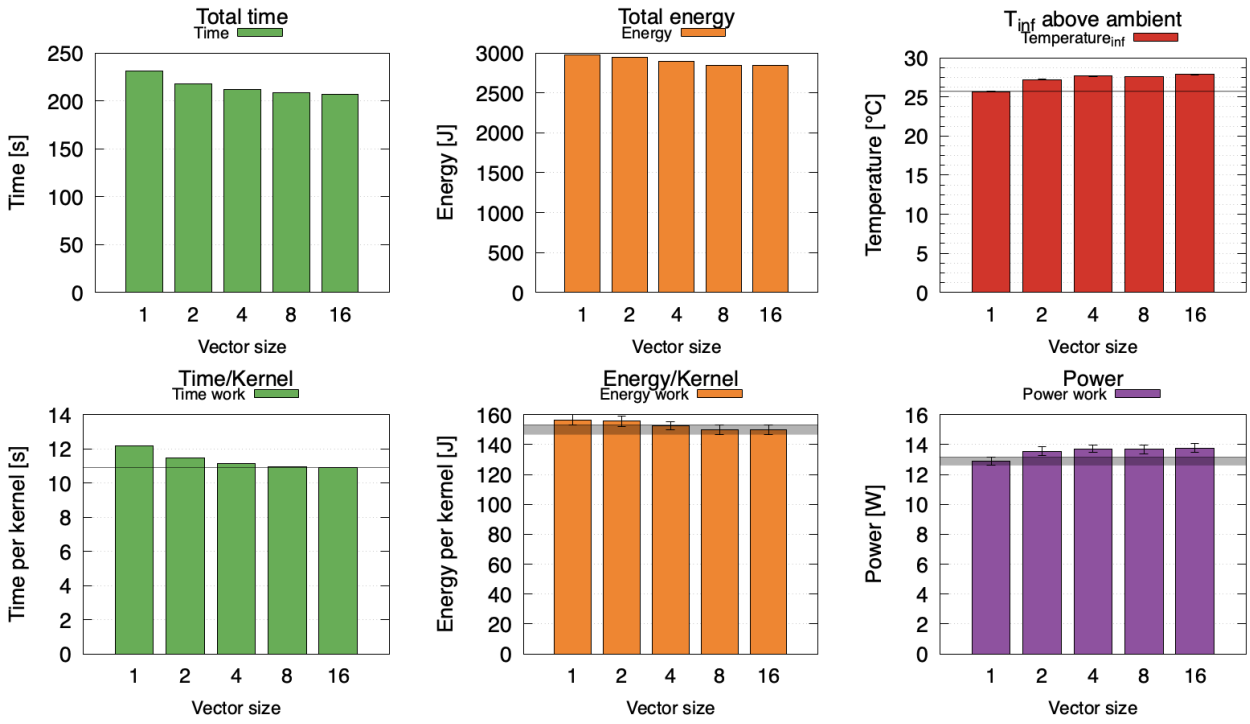


Figure 5.8: Integer compute shows small performance increase with bigger vector sizes due to lower dispatch overhead.

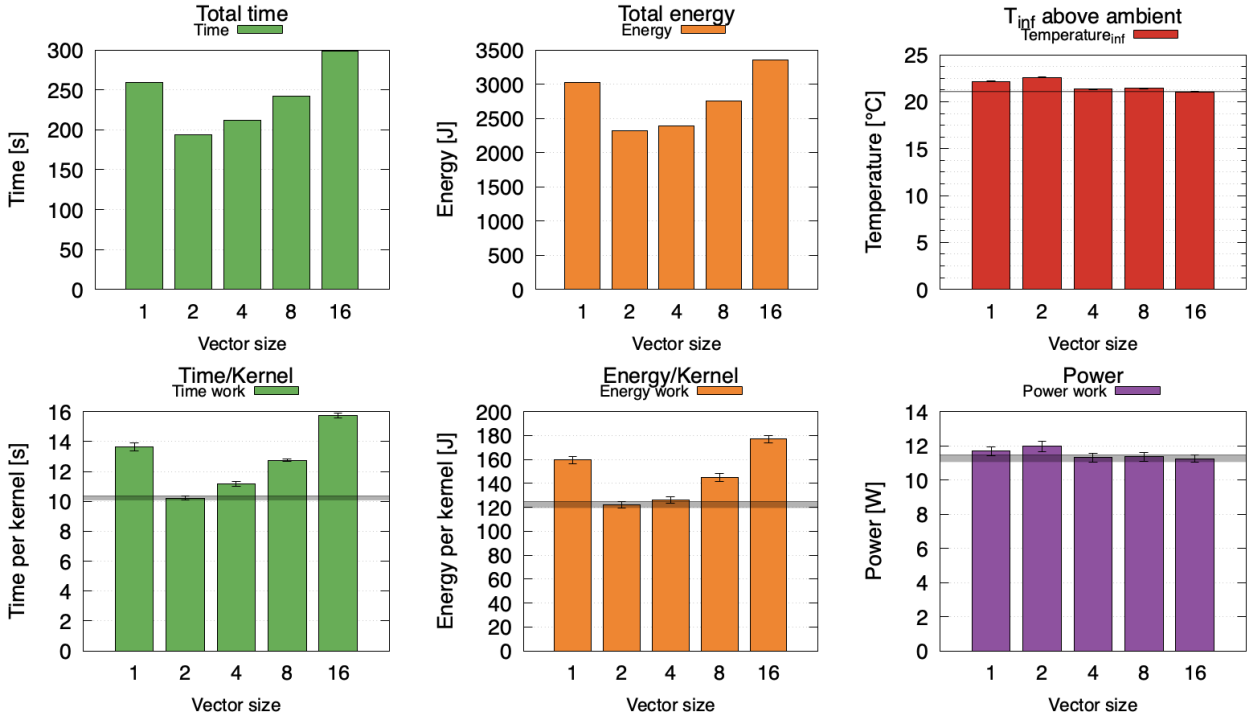


Figure 5.9: In memory bandwidth results provide room for making tradeoffs based on concrete requirements. Vector size 2 has lowest run time, but we can reduce temperature by 5% with 9% increase in run time by setting vector size to 4.

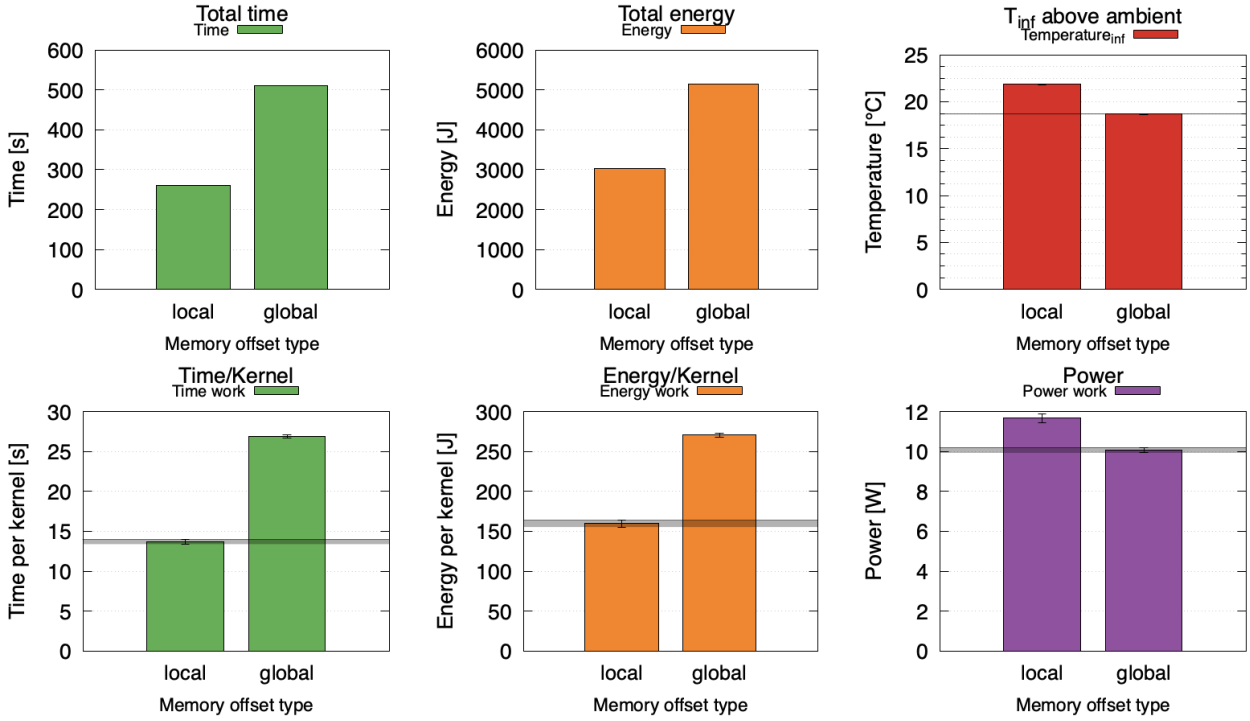


Figure 5.10: Local grouping in memory bandwidth is faster but less power-efficient due to being closer in memory and thus more cache friendly.

### 5.2.5 Graphics API and GPU affinity

We can see in Fig. 5.11 that the dispatch overhead of the Vulkan API is much lower (approximately two times) per-kernel than in OpenCL. The Vulkan API also has less variance in dispatch time, which translates into less variance in energy consumption.

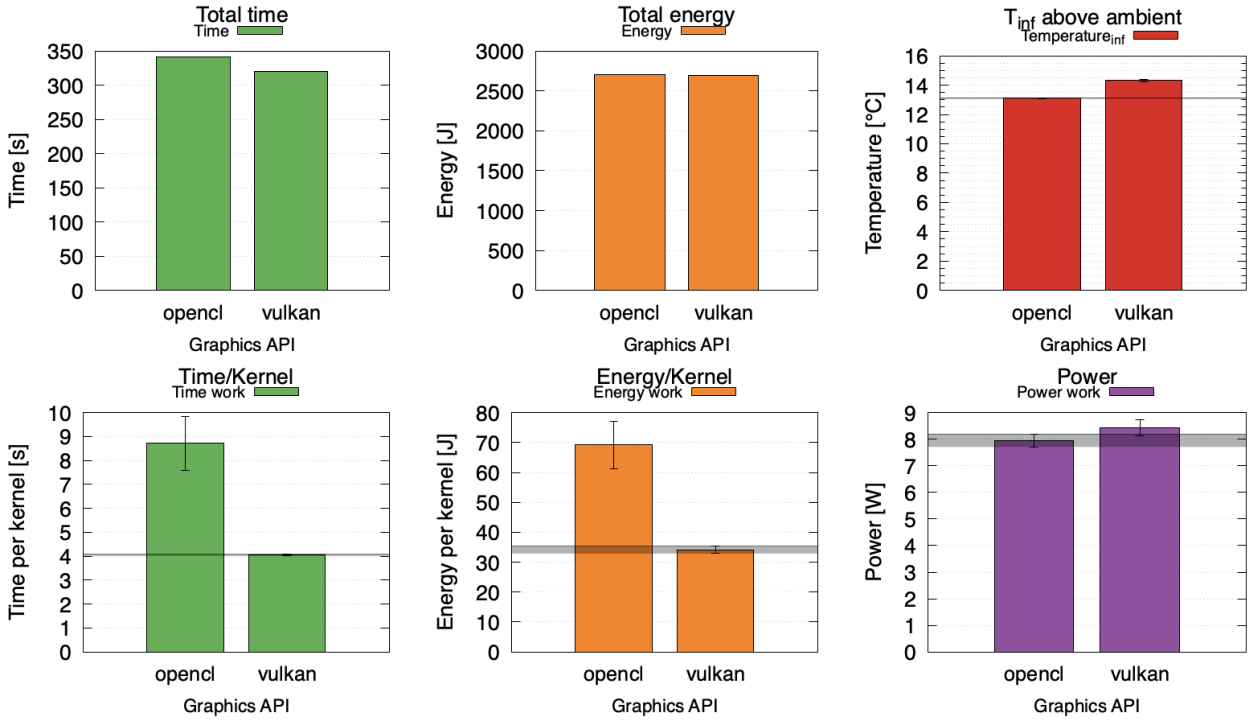


Figure 5.11: Overhead of OpenCL vs Vulakn API. The Vulkan benchmark was run twice as many times to achieve similar total runtime.

Unfortunately as can be seen from Fig. 5.12 Vulkan is not able to utilize both GPU cores. When compared with OpenCL on just one core it appears OpenCL is still slightly more efficient energy-wise.

As for core affinity we can see that using only one core reduces both temperature and power consumption – this is likely due to reduced need for synchronization and multi-core scheduling. We also observe a difference between which GPU core is being utilized – with the `1:1` consuming the same energy but having a lower maximum temperature. Other benchmarks behaved similarly with respect to GPU affinity.

While it appears that while Vulkan provides benefits in performance of the API itself, it is not mature enough with respect to the ecosystem to be viable in production use without much more thorough testing than OpenCL. However, its ongoing adoption might be an indicator of change in this space, under the assumption that critical bugs are fixed.

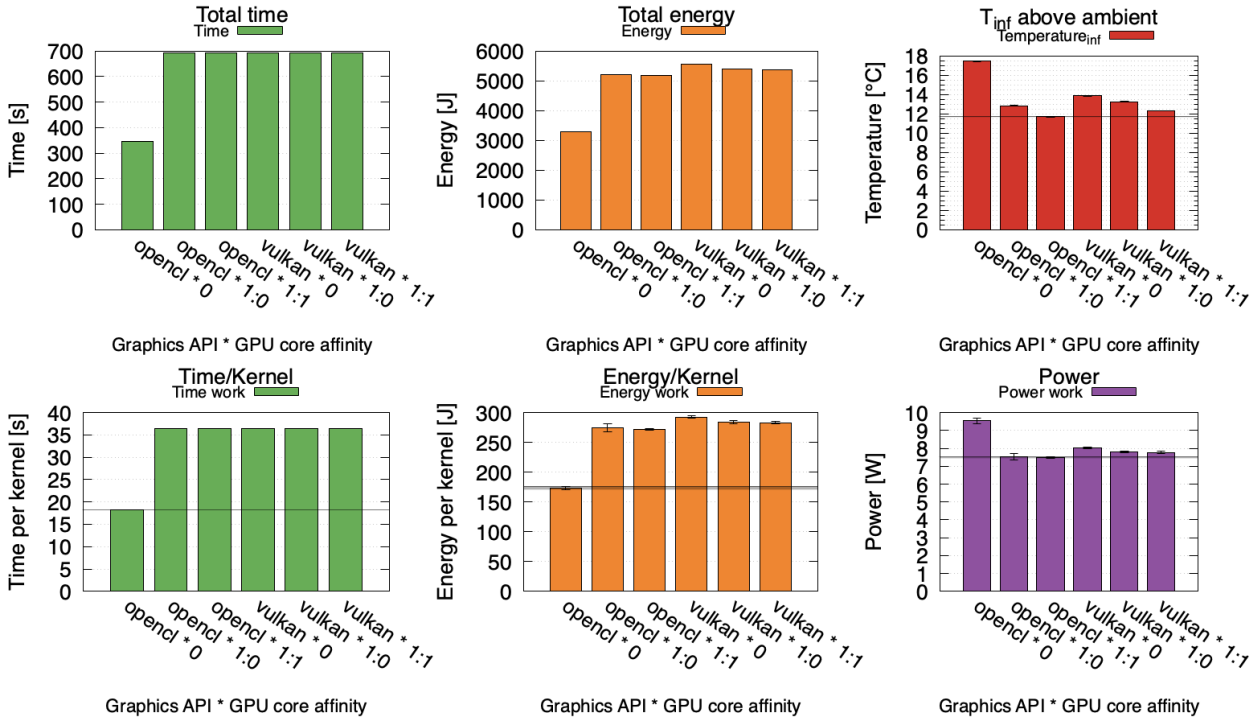


Figure 5.12: Effect of the `VIV_MGPU_AFFINITY` variable and graphics API on the mandelbrot benchmark. The maximum temperature when runnig on the GPU core `1:1` is about 1°C lower than core `1:0`.

### 5.2.6 CPU/GPU work split

In Mandelbrot benchmark we tested whether splitting the workload by the X and Y axis results in different performance. In Fig. 5.13 we can see that the performance is the same in both instances, so we split along Y axis in other measurements, such as Fig. 5.14.

If we explore the effect of disabling CPU cores on CPU/GPU work split (Fig. 5.15) we discover that offloading Mandelbrot in particular yields the best results for all CPUs enabled. This is in contrast with later ADASMark results in Fig. 5.17 where the workload doesn't fully saturate all cores and performance can be improved by utilizing only A72 cores.

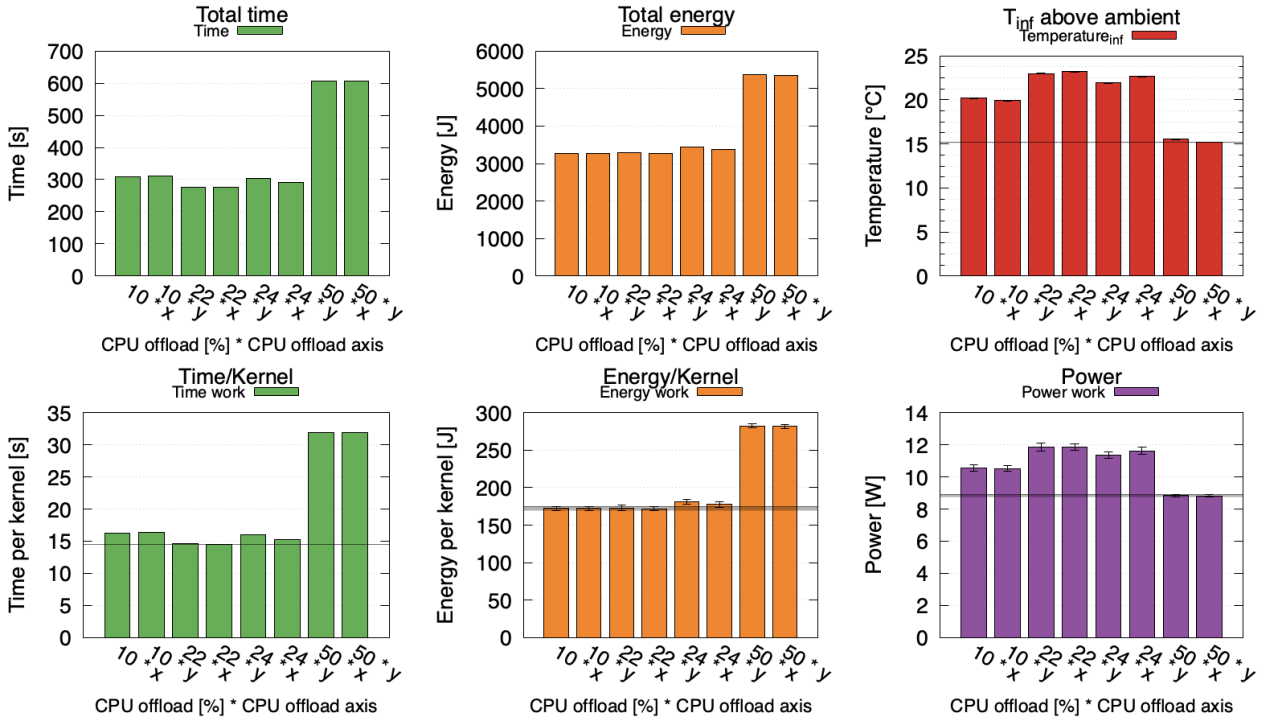


Figure 5.13: In 2d Mandelbrot splitting by X and Y axis results in the same performance.

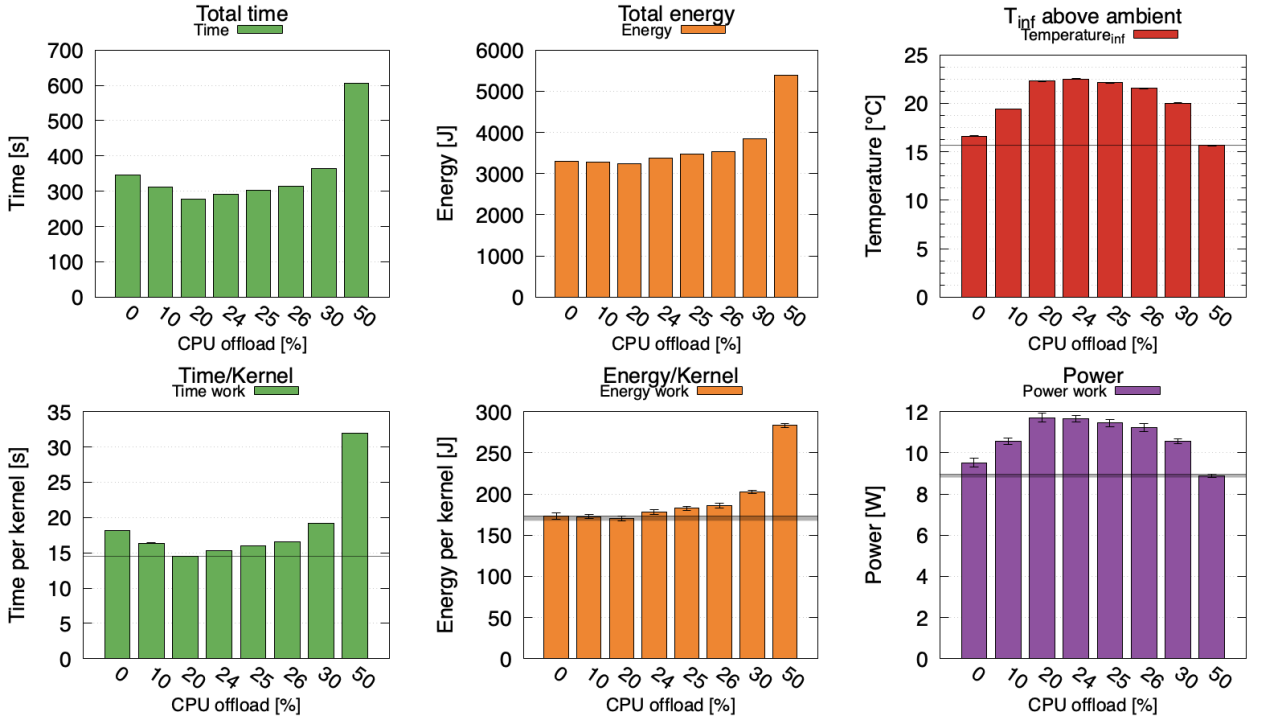


Figure 5.14: By offloading 20% of the mandelbrot work onto the CPU we can achieve 20% reduction in time per kernel with the same energy consumption. However, the maximum temperature increases by 34%.

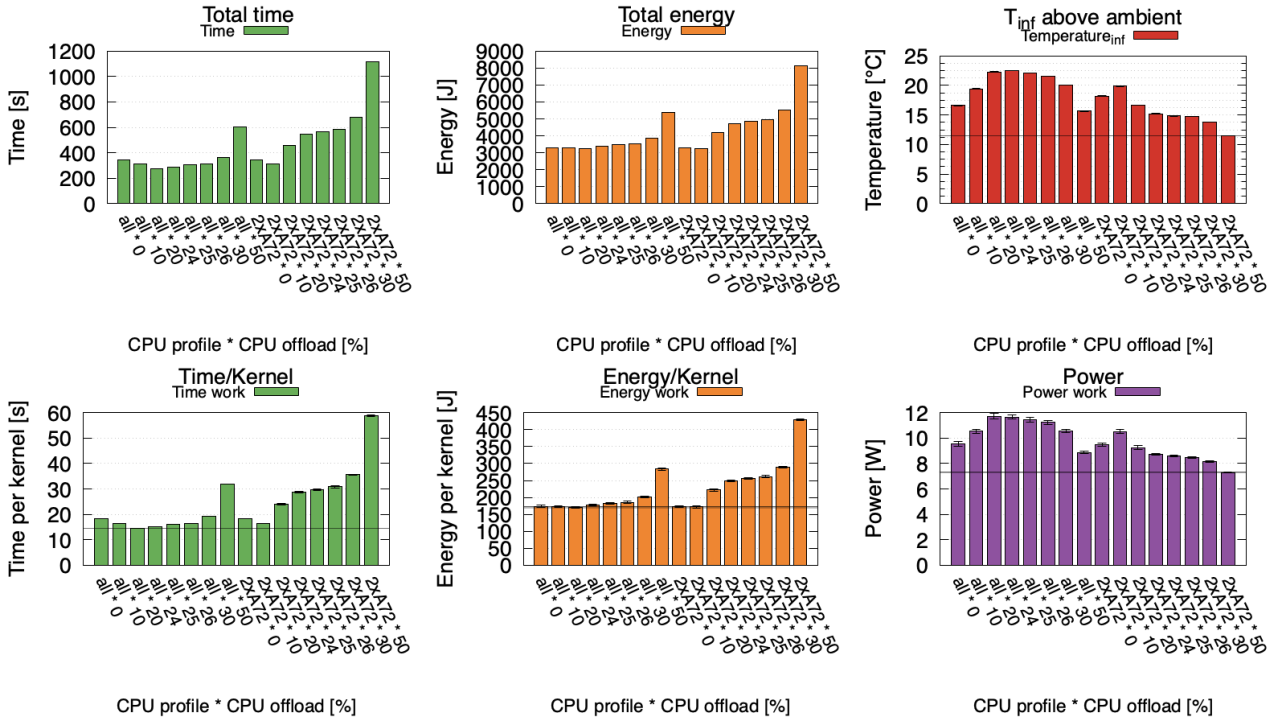


Figure 5.15: Enabling only performance cores during mandelbrot CPU/GPU work split increases run time by 65% while decreasing temperature by 25%.

## 5.3 ADASMark

Finally, in the ADASMark application benchmark we apply findings from previous benchmarks. From combining CPU/GPU work split by running debayer node on CPU and DVFS by setting the GPU to nominal, we can save energy and reduce maximum temperature while having the same run time. See Fig. 5.16.

Due to heterogenity of the CPU package in Fig. 5.17 we can also see that disabling LITTLE cores (or alternatively pinning the process to the big cores) enables all pipelines to perform better, at the cost of higher energy consumption and thus higher maximum temperature. This is due to the workload being scheduled mostly on a single thread which happens to get allocated to the LITTLE cores first. The scheduling could use improvements in this area, but it is unclear whether this is an issue that could be solved in the Linux scheduler or by improving the implementation of hardware drivers. Another possiblity is that ADASMark is an atypical workload with respect to expectations of the implementation.

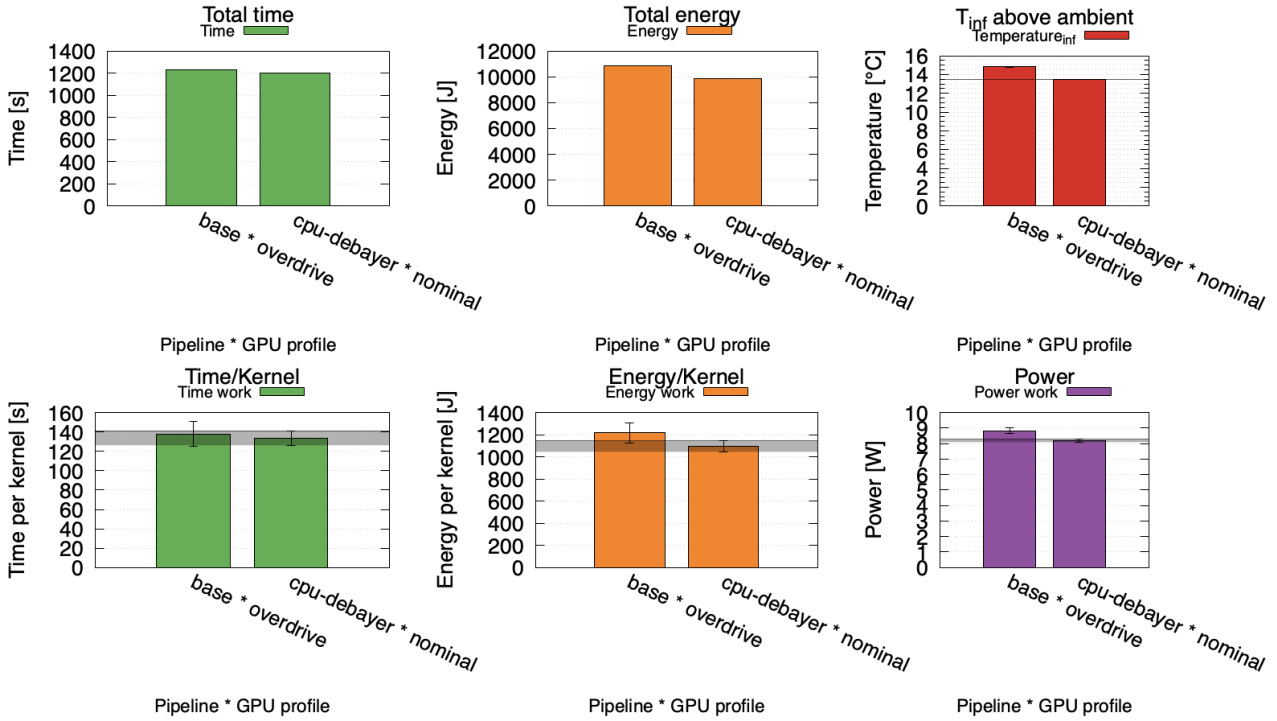


Figure 5.16: Combining CPU/GPU work split and DVFS allows us to reduce maximum temperature noticeably in ADASMark, from 14.8°C to 13.5°C (~9%).

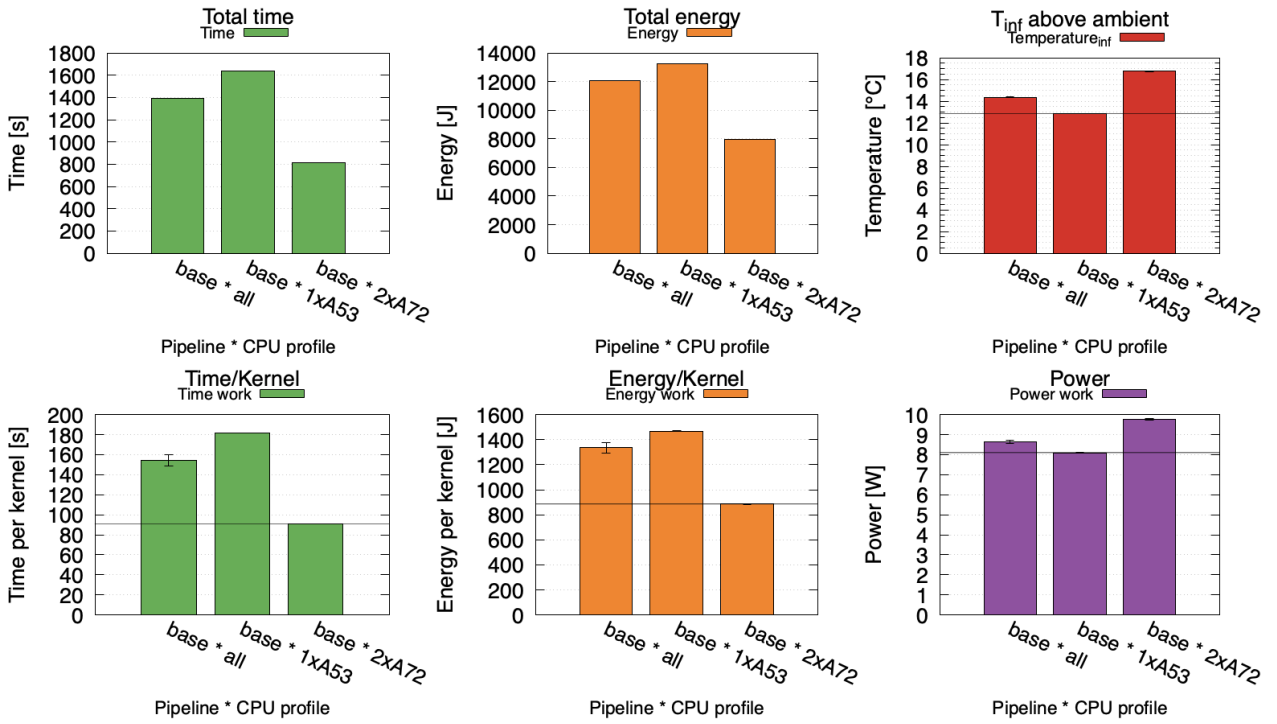


Figure 5.17: Forcing the ADASMark benchmark to run only on the A72 cores improves the performance. We see 41% run time reduction with 17% increase in temperature.

# 6 Conclusion

We create a Yocto layer with new and edited recipes to prepare a Linux distribution with software libraries needed in this work. By utilizing userspace NFS server and novaboot we repeatedly booted the board with prepared operating system. We developed automatization scripts to reproducibly run measurements by executing benchmarks implemented in Rust. In addition to documentation and available source code we used Ghidra to reverse engineer graphics drivers to study their behavior. Using Thermobench framework and hardware testbed we collected temperature and energy data into CSV files. Finally we post-processed this data and visualized it into graphs using Julia scripts with help of Thermobench.jl library.

In results we present system configurations to reduce chip temperature – 1 or 2 °C – while maintaining or only slightly reducing performance. In ADASMark benchmark we reduce maximum temperature by 9% by offloading 25% of work from the GPU to the CPU (using PoCL library) and reducing GPU power using DVFS. We also present a way to decrease run time by 41% with 16% increase in temperature. It remains that optimizing implementation for specific software-hardware vendor/combination is non-trivial and there is no silver bullet for all scenarios, so the general advice is to perform measurements on actual target hardware with the target software stack early and measure regressions or improvements during development.

This can be simplified by using the tooling developed in this work, as described in Sec. 4.2. This tooling was made with the intention to be as flexible as possible and improved as this work was being developed. We believe it is suitable to be used in future work and should be easy to adapt to other platforms and problems. It is released under the MIT license and can be freely modified and redistributed, available at Gitlab [1].

Maintaining an entire operating system distribution is very complex and during our work we have found multiple bugs. Some could be worked around by editing the source code, some were already known and fixed in newer versions and some were in proprietary code and were reported to the appropriate place (see Sec. 4.1 and Sec. 4.3.3). Overall, we attempted to debug and/or report all encountered bugs.

# Bibliography


[1] "thermac-opencl." [Online]. Available: https://gitlab.fel.cvut.cz/lavusedu/thermac-opencl. [Accessed: 13-Jan-2022]

[2] "Thermal-aware Resource Management for Modern Computing Platforms in the Next Generation of Aircraft." [Online]. Available: http://www.cister.isep.ipp.pt/projects/thermac/. [Accessed: 12-Jan-2022]

[3] Hornof, D., "Offline scheduling of the safety-critical tasks within the isolation time-windows," 2021 [Online]. Available: http://hdl.handle.net/10467/95363

[4] Hosseinimotlagh, S. and Kim, H., "Thermal-Aware Servers for Real-Time Tasks on Multi-Core GPU-Integrated Embedded Systems," presented at the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Montreal, QC, Canada, 2019, doi: 10.1109/RTAS.2019.00029.

[5] Lee, Y., Shin, K., and Chwa, H., "pocl: A Performance-Portable OpenCL Implementation," vol. 18, no. 5, pp. 1--25, 2019, doi: 10.1145/3358235.

[6] Lucas, J. and Juurlink, B., "MEMPower: Data-Aware GPU Memory Power Model," vol. 11479, pp. 195--207, 2019, doi: https://doi.org/10.1007/978-3-030-18656-2_15.

[7] "i.MX 8QuadMax/QuadPlus Multisensory Enablement Kit." [Online]. Available: https://web.archive.org/web/20220106134145/https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx-8quadmax-multisensory-enablement-kit-mek:MCIMX8QM-CPU. [Accessed: 06-Jan-2022]

[8] "Cortex-A72." [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a72. [Accessed: 06-Dec-2022]

[9] "Vivante® GPU IP." [Online]. Available: https://www.verisilicon.com/en/IPPortfolio/VivanteGPUIP. [Accessed: 06-Dec-2022]

[10] "U-Boot." [Online]. Available: https://www.denx.de/wiki/U-Boot/. [Accessed: 12-Jan-2022]

[11] "A tool that automates booting of operating systems on target hardware or in qemu." [Online]. Available: https://github.com/wentasah/novaboot. [Accessed: 12-Jan-2022]

[12] "UNFS3 is a user-space implementation of the NFSv3 server specification." [Online]. Available: https://github.com/skoudmar/unfs3. [Accessed: 13-Nov-2022]

[13] "Direct Rendering Manager (DRM)." [Online]. Available: https://dri.freedesktop.org/wiki/DRM/. [Accessed: 13-Nov-2022]

[14] "DRM Architecture by Javier Cantero - Own work, CC BY-SA 4.0." [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=38185134. [Accessed: 13-Nov-2022]

[15] "The Mesa 3D Graphics Library." [Online]. Available: https://mesa3d.org/. [Accessed: 13-Nov-2022]

[16] "OpenCL Overview." [Online]. Available: https://www.khronos.org/opencl/. [Accessed: 06-Jan-2022]

[17] "OpenCL ICD Loader." [Online]. Available: https://github.com/KhronosGroup/OpenCL-ICD-Loader. [Accessed: 12-Jan-2022]

[18] Jääskeläinen, P., de La Lama, C.S., Schnetter, E., *et al.*, "pocl: A Performance-Portable OpenCL Implementation," vol. 43, no. 5, pp. 752--785, Oct. 2015, doi: 10.1007/s10766-014-0320-y.

[19] "The LLVM Compiler Infrastructure." [Online]. Available: https://llvm.org/. [Accessed: 13-Jan-2022]

[20] "Cross platform 3D Graphics." [Online]. Available: https://www.vulkan.org/. [Accessed: 12-Jan-2022]

[21] "Vulkayes - rust wrapper over Vulkan." [Online]. Available: https://github.com/vulkayes. [Accessed: 12-Jan-2022]

[22] "Rust Programming Language." [Online]. Available: https://www.rust-lang.org/. [Accessed: 12-Jan-2022]

[23] Lavuš, E., "Implementation of rendering system in Rust," 2020 [Online]. Available: http://hdl.handle.net/10467/87753

[24] "Yocto Project." [Online]. Available: https://www.yoctoproject.org/. [Accessed: 06-Jan-2022]

[25] "Yocto Project diagram." [Online]. Available: https://www.yoctoproject.org/software-overview/. [Accessed: 06-Dec-2022]

[26] "i.MX Release Manifest." [Online]. Available: https://source.codeaurora.org/external/imx/imx-manifest. [Accessed: 06-Jan-2022]

[27] "The ADASMark™ Benchmark." [Online]. Available: https://www.eembc.org/adasmark/. [Accessed: 13-Nov-2022]

[28] "Ghidra - A software reverse engineering (SRE) suite of tools." [Online]. Available: https://ghidra-sre.org/. [Accessed: 13-Nov-2022]

[29] "imx8qmmek + OpenCL linux kernel panics." [Online]. Available: https://community.nxp.com/t5/i-MX-Graphics/imx8qmmek-OpenCL-linux-kernel-panics/m-p/1386587. [Accessed: 06-Jan-2023]

[30] "Rust bindgen." [Online]. Available: https://github.com/rust-lang/rust-bindgen. [Accessed: 12-Jan-2022]

[31] "OpenCL™ API Headers." [Online]. Available: https://github.com/KhronosGroup/OpenCL-Headers. [Accessed: 12-Jan-2022]

[32] "GPU affinity and Vulkan." [Online]. Available: https://community.nxp.com/t5/i-MX-Graphics/GPU-affinity-and-Vulkan/m-p/1510231. [Accessed: 06-Jan-2023]

[33] "PoCL Usage." [Online]. Available: http://www.portablecl.org/docs/html/using.html. [Accessed: 06-Jan-2023]

[34] "clpeak - A tool which profiles OpenCL devices to find their peak capacities." [Online]. Available: https://github.com/krrishnarraj/clpeak. [Accessed: 13-Jun-2022]

# Contents of the included data disk

The included data is a copy of the repository which is also avaialble online. It contains its own README.md file which documents specific details. An overview of the contents of the repository:

The `docs` directory contains guides and code examples made available by the NXP, ADASMark, Thermac and OpenCL.

The `system` directory contains bitbake recipes used in this work, as well as scripts used to boot the system using the novaboot setup. The boot proces requires an existing NFS server (possibly using userspace NFS), FTP server and a novaboot server.

The `experiments` directory contains all benchmark implementations and configurations utilized in this work. It contains the low-level OpenCL wrapper library, experiment implementations including OpenCL kernels and Vulkan shaders, ADASMark pipeline configurations and well as the measurement runner. The measurement runner directory also contains the JSON definition file.

Finally, the `results` directory contains both the measured results (mostly csv) as well as Julia scripts used to load, process and graph these results.

```
$ exa --tree --level 3 --only-dirs
.
├── docs
│   └── c_example
├── experiments
│   ├── adasmark
│   ├── experiments
│   │   ├── assets
│   │   └── src
│   ├── libs
│   │   └── aarch64-linux-gnu
│   ├── opencl-ffi
│   │   ├── OpenCL-Headers
│   │   └── src
│   ├── opencl-lib
│   │   └── src
│   └── runner
├── results
│   ├── drawings
│   └── thermobench
│       └── ...
└── system
    ├── etnaviv
    ├── meta-thermac
    │   ├── conf
    │   ├── recipes-core
    │   ├── recipes-graphics
    │   └── recipes-kernel
    └── misc
```