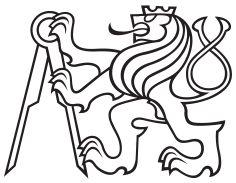
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Tools for the Development of Advanced Thermal Management Techniques for Future Safety-Critical Embedded Systems

Tibor Rózsa

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Open Informatics
Subfield: Computer Vision and Image Processing
August 2020

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rózsa Tibor** Personal ID number: **452919**

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

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Computer Vision and Image Processing**

II. Master's thesis details

Master's thesis title in English:

Tools for the Development of Advanced Thermal Management Techniques for Future Safety-Critical Embedded Systems

Master's thesis title in Czech:

Nástroje pro vývoj pokročilých technik řízení teploty pro safety-critical vestavěné systémy

Guidelines:

1. Make yourself familiar with multi-core heterogeneous (CPU+GPU) chips for industrial applications such as NXP i.MX8.
2. Develop a tool to measure temperature at various places on the chip and/or board using a thermal camera (perhaps using provided SDK). The measurement points should remain the same even after small board/camera movements.
3. Develop a benchmarking tool (SW) for measuring thermal properties of the HW platform running a Linuxbased OS under various workloads.
4. Use the developed tools to determine heat sources on the chip for different workloads (both micro- and macro-benchmarks).
5. Propose a method for reducing chip temperatures during execution of the following workloads: a) software 3D renderer and b) image processing algorithms (object tracking). Use the developed benchmarks and tools from previous points to evaluate effectiveness of the proposed method.
6. Document all findings.

Bibliography / sources:

[1] Hartley, R. and Zisserman, A. Multiple View Geometry in Computer Vision. Cambridge University Press, 2nd ed, 2003.
[2] K. Dev, I. Paul, W. Huang, Y. Eckert, W. Burleson, and S. Reda, "Implications of Integrated CPU-GPU Processors on Thermal and Power Management Techniques," arXiv:1808.09651 [cs], Aug. 2018.

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: **17.01.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Michal Sojka, Ph.D. Supervisor's signature	doc. Ing. Tomáš Svoboda, Ph.D. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor, Michal Sojka for his invaluable guidance throughout my work in CIIRC; other members of the CIIRC THERMAC team (Ondřej Benedikt, Joel Matějka, František Fladung, Alex Barinov) for their insight and contributions to the tools described in this thesis; and Ondřej Drbohlav for his valuable advice in image processing.

Declaration

I declare that I wrote the presented thesis on my own and that I cited all the used information sources in compliance with the methodical instructions about the ethical principles for writing an academic thesis.

In Prague, 14. August 2020

Abstract

Software-based temperature reduction methods show great potential for small aircraft avionics computing platforms by allowing improved dependability, performance and reduction in size and weight without increasing hardware costs. To evaluate such methods, we present a pair of tools for recording and processing data from temperature sensors and a thermal camera during the execution of various workloads. These tools are then used to determine locations of on-chip heat sources and to propose methods for reducing chip temperature. The tools meet their requirements and are successfully used for the evaluation of temperature reduction methods.

Keywords: temperature measurement, on-chip sensor, thermal camera, heat source, thermal management

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Softwarové metody snižování teploty ukazují velký potenciál pro výpočetní platformy malých letadel pro avioniku tím, že umožňují zvýšenou spolehlivost, výkon a zmenšení velikosti a hmotnosti bez zvýšení nákladů na hardware. Pro vyhodnocení těchto metod uvádíme dvojici nástrojů pro záznam a zpracování dat z teplotních senzorů a z termální kamery při různých pracovních zatížení.. Tyto nástroje jsou pak použité k lokalizaci zdrojů tepla na čipu a k navrhování metod pro snižování teploty čipů. Nástroje splňují jejich požadavky a jsou úspěšně použité pro vyhodnocení metod snižování teploty.

Klíčová slova: měření teploty, senzor na chipu, termokamera, zdroj tepla, tepelné řízení

Contents

Figures

Tables

Chapter 1

Introduction

Many modern computing platforms in safety-critical domains such as aviation are based on **Multiprocessor System-on-Chip (MPSoC)**. Such platforms guarantee high performance as long as the chip stays in a strict thermal range, which is achieved through either active cooling (usually implemented by forcing airflow through a CPU fan) or passive cooling (implemented by installing a heat sink).

Active cooling often complicates mechanical design and may not be viable due to limited airflow. Passive cooling is less effective and using larger heat sinks may significantly increase weight and size. Thus, there is a need for complementary methods to reduce the peak temperatures in MPSoCs chips.

The work described in this thesis is part of the THERMAC project. The main goal of the project is to develop software-based methods to reduce the operating temperature of MPSoC platforms for avionics applications. The decreased temperature improves dependability, computing performance, and reduces size and weight of electronics due to relaxed dissipation requirements. It also allows the integration of a higher number of functionalities on the same computing platform.

Specifically, the project aims to:

- Reduce operating temperature by 20% for equivalent guaranteed performance
- Increase guaranteed performance by 30% for an equivalent thermal profile

The main contribution of this thesis is the development of measurement tools, which can then be used to evaluate the temperature reduction methods:

- A temperature measurement tool running on an embedded platform with a low performance impact – Thermobench (chapter 3)
- Benchmarks measuring thermal properties – Thermobench benchmarks (section 3.3)
- A temperature measurement tool based on a thermographic camera using point tracking – Thermocam-PCB (chapter 5)

The Thermocam-PCB tool is a major focus of this thesis, as long-term precise tracking of points on thermal image stream proves to be a nontrivial image processing problem, especially when the tracking has to be stable through a wide range of temperatures. Chapter 4 provides a background for tracking methods in image processing and is used as a reference in chapter 5, which details the Thermocam-PCB tool.

We then use these tools to detect on-chip heat sources and evaluate temperature reduction methods, described in chapters 6 and 7. The rest of the THERMAC team also uses the developed tools for their experiments – these will not be detailed in the thesis.

Chapter 2

Hardware platforms

2.1 Target hardware platforms

As part of the THERMAC project described in chapter 1, we chose to determine the thermal properties of two specific platforms: **i.MX8QMMEK – NXP i.MX 8QuadMax Multisensory Enablement Kit** and **NVIDIA Jetson TX2 Developer Kit** (Figures 2.1a and 2.1b).

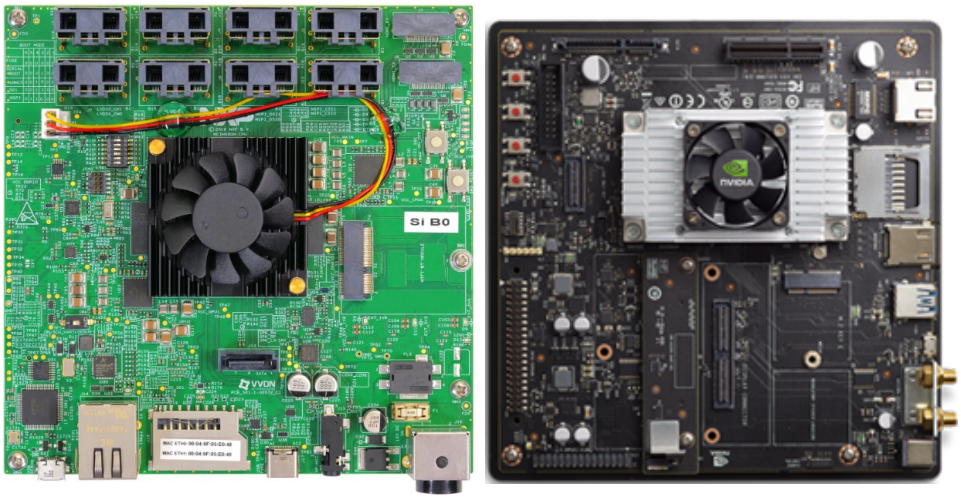
Both hybrid CPU-GPU platforms have high computational performance compared to popular embedded platforms (e.g. a Raspberry Pi) – their technical specifications are shown in Table 2.1. They are designed for high-bandwidth embedded applications, such as advanced video/audio processing in industrial or automotive environments.

2.2 Thermographic camera & processing hardware

We needed a separate platform to process the output of the **Workswell Infrared Camera (WIC)** (Figure 2.2a) using the WIC SDK [Jer17]. We chose the **Minnowboard Turbot** embedded platform (Figure 2.2b), as it can run Ubuntu 16.04 x64, which is required by the free version of the WIC SDK. The specifications for the WIC are in Table 2.2, for the Turbot board they are in Table 2.3.

	NXP i.MX8QMMEK	NVIDIA Jetson TX2 Developer Kit
Chip	NXP i.MX8	NVIDIA Parker Series SoC
CPU cores	2x Cortex-A72 @ 1.6 GHz	4x ARM Cortex-A57 @ 1.4 GHz
	4x Cortex-A53 @ 1.2 GHz	2x NVIDIA Denver 2 64b @ 1.1 GHz
	2x Cortex-M4 @ 266 MHz	
GPU	4-core NXP @ 800 MHz	256-core NVIDIA Pascal @ 1.3 GHz
Memory	3GB LPDDR4 64b @ 1.6 Ghz	8GB LPDDR4 128b @ 1.8 Ghz
Storage	32GB eMMC 5.1	

Table 2.1: NXP i.MX8QMMEK & NVIDIA Jetson TX2 Developer Kit technical specifications [NVI20][NXP20]



(a) : NXP i.MX 8QuadMax Multisensory Enablement Kit[NXP20]

(b) : NVIDIA Jetson TX2 Developer Kit [NVI20]

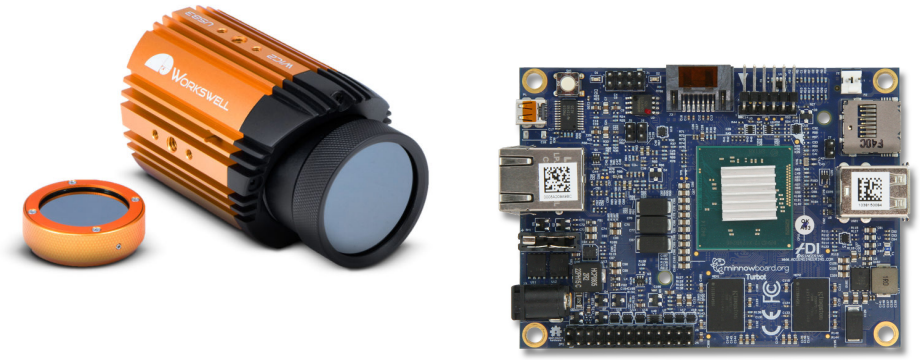
Figure 2.1: Embedded platforms used by Thermobench

Resolution	336x256
Sensitivity	0.03 °C
Accuracy	±2% or ±2 °C
Temperature range	-25 °C . . . +150 °C
	-40 °C . . . +550 °C
	50 °C . . . 1000 °C with external filter
Frame rate	9 Hz
Spectral range	7.5 – 13.5 µm
Communication	USB3 or GigE

Table 2.2: WIC technical specifications [Wor20]

CPU cores	2x Intel Atom E3826 @ 1.46 GHz
Memory	2GB DDR3L 1.0 GHz
Storage	4GB (Micro SD slot)

Table 2.3: MinnowBoard Turbot Board technical specifications [Rub20]



(a) : Workswell Infrared Camera [Wor20] **(b) :** MinnowBoard Turbot Board [Rub20]

Figure 2.2: Hardware used by Thermocam-PCB

Chapter 3

Thermobench and related tools

Thermobench is a measurement tool for running benchmark applications and reading and recording data from on-chip sensors(temperature, frequency, etc.) while the benchmark runs. It was created to be used in the THERMAC project described in chapter 1, for evaluating temperature reduction methods.

I implemented the base functionality and some simple features for the tool; it was later improved by the THERMAC team, who added several additional features, as well as improvements to code readability, maintainability and functionality. Some notable features are shown in Table 3.1.

As well as adding features to the tool itself, the THERMAC team developed additional tools for processing the output (CSV files) of Thermobench and benchmarks to run on the tool – these are detailed in sections 3.2 and 3.3.

3.1 Thermobench

The following section details the functionality and implementation details of the Thermobench command line tool.

Author	Feature
Tibor Rózsa	Wait for device to cool down to given temperature
	Calculate and log CPU usage
	Time limit for benchmark
	Log benchmark stdout to CSV
	Set column name for CSV and parse benchmark stdout for col_name=value lines, record values into column
Michal Sojka	Controllable fan speed
	Execute additional command and parse its stdout for specified CSV columns (STR=val format)

Table 3.1: Features of the Thermobench tool

3.1.1 Requirements

The main requirements for the Thermobench benchmarking tool are to:

1. Run benchmark applications
2. Measure and record the output of various sensors (temperature, frequency, etc.) during the benchmark run
3. Save measurements in a common format – comma-separated values (CSV)
4. Have negligible computational load (to influence measurements as little as possible)

3.1.2 Design and implementation

Thermobench is designed to be an easy to use, lightweight command line tool. It is written in C++, uses the Meson build system and has no external dependencies. Its build system allows to simply cross compile it and Thermobench benchmarks for the ARM64 architecture.

An example of a very simple use case of the tool is:

```
build/thermobench \
    --sensor=/sys/devices/virtual/thermal/thermal_zone{0..3}/temp \
    benchmarks/CPU/instr/read
```

Here Thermobench runs the *read* benchmark located at benchmarks/CPU/instr, reads the internal temperature sensors while the benchmark runs, and saves them into a CSV file with the name "read.csv" to the current directory.

A more complex example is:

```
build/src/thermobench --sensors_file=src/sensors.imx8 --wait=40 \
    --period=500 --cpu-usage --time=600 \
    build/benchmarks/CPU/instr/simd_int32_mul
```

Here Thermobench reads the paths to the internal sensors of the i.MX8 platform from a sensor file (see command line reference), waits until the CPU temperature (first path in the sensor file) drops to 40 °C, runs the arithmetic benchmark for SIMD 32bit integer multiplication, and records sensor values and CPU usage every 500 ms for 10 minutes.

The command line reference for the tool is shown below:

```
Usage: thermobench [OPTION...] [--] COMMAND...
Runs a benchmark COMMAND and stores the values from temperature (and other)
sensors in a .csv file.

  -c, --column=STR           Add column to CSV populated by STR=val lines from
                             COMMAND stdout
  -e, --exec=[(COL[,...])]CMD   Execute CMD (in addition to COMMAND) and store
                             its stdout in relevant CSV columns as specified by
                             COL. If COL ends with '=', such as 'KEY=', store
                             the rest of stdout lines starting with KEY= in
                             column KEY. Otherwise all non-matching lines will
                             be stored in column COL. If no COL is specified,
                             first word of CMD is used as COL specification.
                             Example: --exec '(amb1=,amb2=,amb_other) ssh
                             ambient@turbot read_temp'
  -E, --exec-wait            Wait for --exec processes to finish. Do not kill
                             them (useful for testing).
  -f, --fan-cmd=CMD          Command to control the fan. The command is invoked
                             as 'CMD <speed>', where <speed> is a number
                             between 0 and 1. Zero means the fan is off, one
                             means full speed.
  -F, --fan-on[=SPEED]       Set the fan speed while running COMMAND. If SPEED
                             is not given, it defaults to '1'.
```

```
  -l, --stdout                Log COMMAND's stdout to CSV
  -n, --name=NAME             Basename of the .csv file
  -o, --output_dir=DIR        Where to create output .csv file
  -O, --output=FILE           The name of output CSV file (overrides -o and -n).
                              Hyphen (-) means standard output
  -p, --period=TIME [ms]      Period of reading the sensors
  -s, --sensors_file=FILE     Definition of sensors to use. Each line of the
                              FILE contains either SPEC as in -S or, when the
                              line starts with '!', the rest is interpreted as
                              an argument to --exec. When no sensors are
                              specified via -s or -S, all available thermal
                              zones are added automatically.
  -S, --sensor=SPEC           Add a sensor to the list of used sensors. SPEC is
                              FILE [NAME [UNIT]]. FILE is typically something
                              like
                              /sys/devices/virtual/thermal/thermal_zone0/temp
  -t, --time=SECONDS          Terminate the COMMAND after this time
  -u, --cpu-usage             Calculate and log CPU usage.
  -v, --verbose               Print progress information to stderr.
  -w, --wait=TEMP [°C]        Wait for the temperature reported by the first
                              configured sensor to be less or equal to TEMP
                              before running the COMMAND. Wait timeout is given
                              by --wait-timeout.
  -W, --wait-timeout=SECS     Timeout in seconds for cool-down waiting (default:
                              600).
  -?, --help                  Give this help list
      --usage                 Give a short usage message
  -V, --version               Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Besides reading and storing the temperatures, values reported by the benchmark
COMMAND via its stdout can be stored in the .csv file too. This must be
explicitly enabled by -c or -l options.

Report bugs to https://github.com/CTU-IIG/thermobench/issues.
```

3.1.3 Results

Thermobench satisfies all requirements established in section 3.1.1. It is capable of running benchmark applications, reading the on-chip sensor data and storing it into a CSV file.

In the current implementation, a recording period of 1s is sufficient to ensure a negligible computational load. This resolution is usually sufficient, as chip temperature changes are generally fairly slow, on the scale of minutes.

3.2 Thermobench grapher tools

The Thermac team developed several tools for processing the CSV files generated by Thermobench:

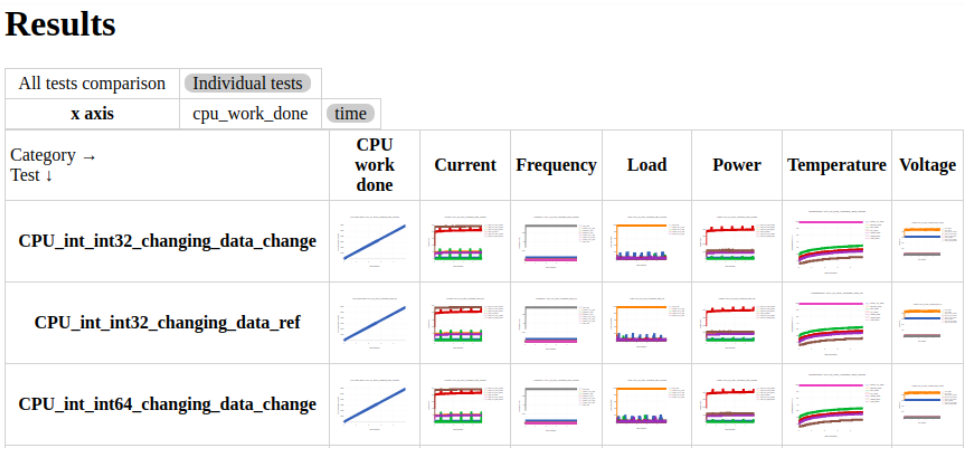
1. **Thermobench report generator** (by Tibor Rózsa)

   Command line tools (**graph** and **report generator**) for automatically creating reports summarizing a large number of benchmark runs. These reports contain plots for different sensor types (temperature, frequency, voltage, etc.) for each benchmark run, which are useful to have at hand when required, but aren't necessarily important for analysis.

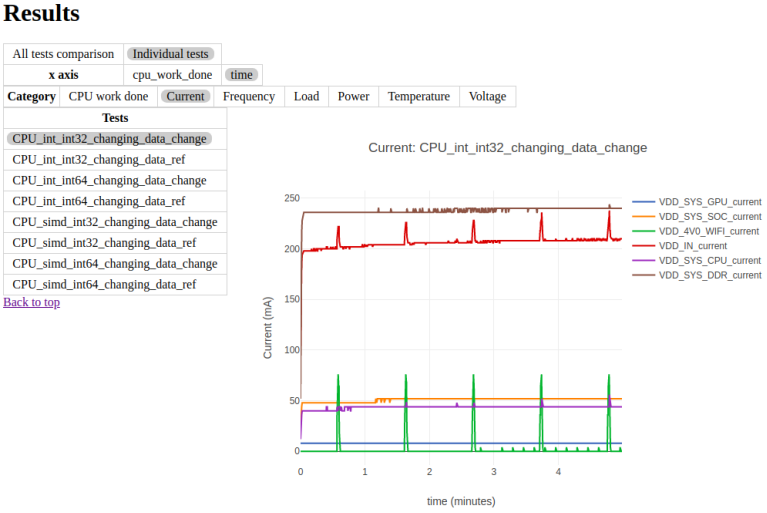
   The tools unfortunately cannot be used to overlay graphs of different units (e.g. temperature and frequency) when they are of different orders of magnitude.

   Graph generator
   As default, if supplied with more than 1 CSV file, the grapher plots all columns that the CSV files have in common in separate graphs. Additional useful options:

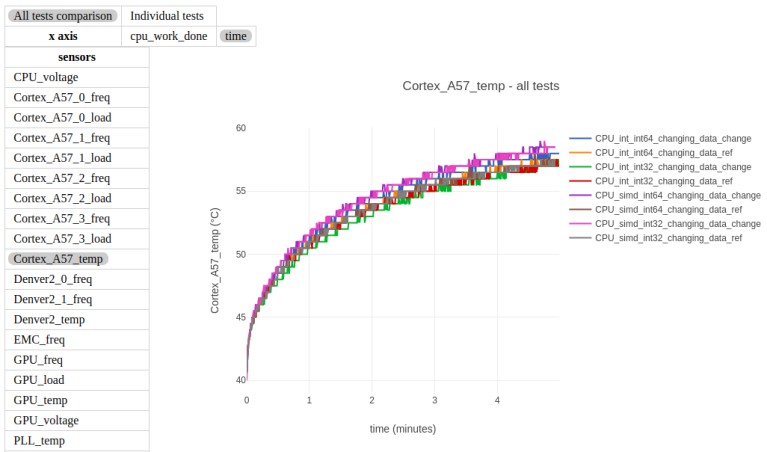
   - Plot only some columns across files (not all of them)
   - Group columns in a single CSV into graph by search strings
     Example: Group all columns containing string "temp" into a single graph with Temperature on the Y-axis
   - Select column(s) for X-axis, plot all other columns with these X-axes



(a) : Overview – table of plots from individual benchmark runs



(b) : Detailed view of benchmark run – grouped together by sensor type (current, temperature, etc.)



(c) : Comparison of CSV columns across all benchmark runs

Figure 3.1: 3 views of HTML generated from the report generator tool

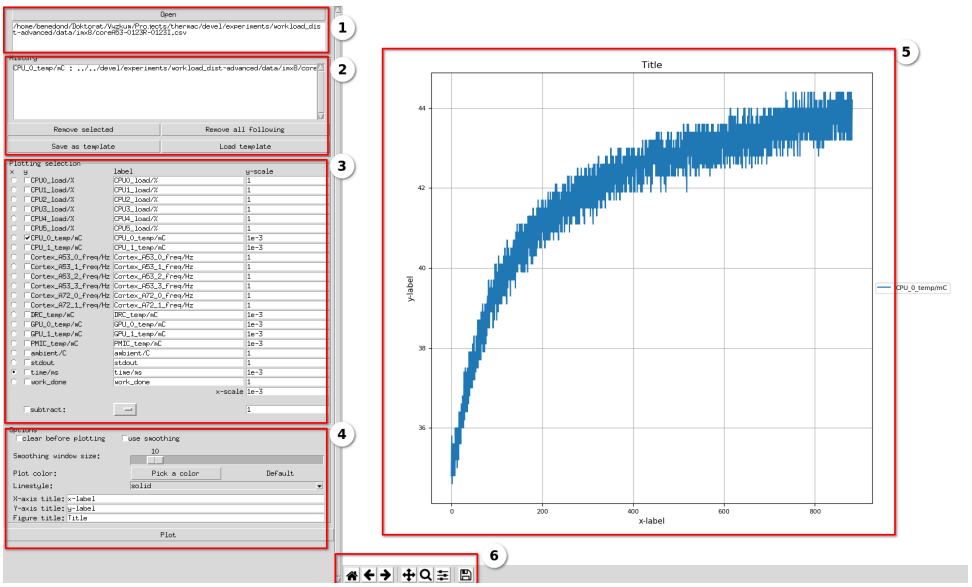


Figure 3.2: GUI of the Thermobench Data Visualizer

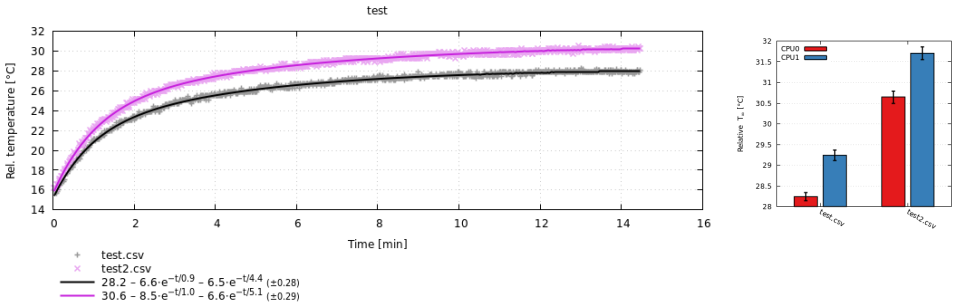
Report generator
Compiles generated graphs into a html page with navigation, with 3 main views:

- Overview – Table of plots, each row contains plots grouping together columns with the same unit in each CSV file (Figure 3.1a)
- Detailed view of plots from overview (Figure 3.1b)
- List consisting of plots grouping together columns of the same name across all CSV files (Figure 3.1c)

2. **Thermobench data visualizer** (by Ondřej Benedikt)
Graphical tool to quickly select columns and create plots from CSV files. This tool is useful for quick, manual construction of graphs.

Figure 3.2 shows the GUI of the program, with the following functions:

Number	Function	Description
1	File selection	Select csv file from file browser
2	Plotting history	History of individual plotted lines – remove lines or save plot template
3	Data-plotting selection	Select x axis column, y axis columns, label and scaling factor for data Can subtract another column from data
4	Plotting options	Clear plot, add smoothing, add plot title, add axis titles, change line style and color
5	Figure	Display of the current plot
6	Toolbar	Save plot to a `.png` file



(a) : `multi_fit` **(b) :** `plot_Tinf`

Figure 3.3: Plots from Thermobench.jl functions

3. **Thermobench.jl** (by Michal Sojka)
Plotting and data processing package based on the Julia [BEKS17] language. Useful functions:

`multi_fit`
In many measurements, the benchmark applies a continuous load to the CPU/GPU for the entire duration of the experiment. The resulting data can be fitted fairly well to a thermal model represented as a step response of an n^{th} order linear dynamic system. `multi_fit` fits the thermal model to the temperature data and plots both the data and the resulting equation – see Figure 3.3a.

Useful parameters include:

- `subtract` – subtract a column (such as ambient temperature) from the data
- `use_measurements` – produce results with confidence intervals

`plot_Tinf`
When comparing benchmark runs with the aforementioned thermal model, we are often only interested in the temperatures at time infinity. `plot_Tinf` visualizes these as a bar chart with error bars – see Figure 3.3b.

3.3 Thermobench benchmarks

To test the thermal properties of our hardware platforms (see section 2.1), we use a combination of standard and custom benchmarks to run our experiments. We wrote custom benchmarks mainly for simple workloads – pure arithmetic for an instruction-dependent workload with no memory access; memory-heavy to to compare the effect of accesses to various cache levels and the main memory. External benchmark sets are mainly used to simulate real-world embedded applications.

- CPU
  - Instruction benchmarks
    - ALU – 32/64bit, float/int, ADD/MUL/MADD/DIV
    - SIMD – 8/16/32/64bit, float/int, ADD/MUL/MADD/DIV
  - Membench – read/write memory benchmark (by Michal Sojka)
  - CoreMark & CoreMark PRO benchmark set (by EEMBC)
  - TACLe-bench – embedded benchmarks for timing analysis (by [FAH+16])
- GPU
  - CUDA instruction benchmarks – 16/32/64bit, float/int, ADD/MUL/DIV
  - OpenCL – memory and compute benchmarks (by František Fladung)

3.4 Thermobench experiments

Collectively the THERMAC team did a large number experiments on both the NXP i.MX8 and NVIDIA TX2 platforms. This section will only summarize a few of my own experiments which had relevant results.

3.4.1 Experiment setup

The goal of the following experiments is to investigate the effect of different kinds of arithmetic instructions on chip temperature.

The arithmetic benchmarks we use consist of 1024 instructions of the same type (int32 add, double div, etc.) written in ARM assembly, nested in an inline function which is called in a loop. This way the overwhelming majority of the instructions in the code are exactly the ones we want to test, not jumps or iteration in the for loop.

The topics and setup for experiments are the following:

1. **SIMD vs ALU execution**

   All ARM processors on our hardware platforms (see chapter 2) contain a NEON module for execution of SIMD code. Although SIMD code can perform notably more operations per time unit than sequential code, it is worth testing whether it is actually worth using from a thermal perspective.

2. **Arithmetic instruction comparison – ADD, MUL, MADD, DIV**

   This experiment investigates whether the arithmetic instructions ADD, MUL, MADD, and DIV have differing effects on chip temperature.

3. **Multiply + Add vs MADD**

   Given the differences in the thermal profiles of various instructions, it is worth examining whether it is useful to use combined instructions such as MADD from a thermal perspective.

   I compare 2 benchmarks, one with 1024 MADD and one with 512 MUL and 512 ADD instructions. When plotting the number of MADDs executed, I count them as 2 instructions worth of work to keep the comparison fair.

4. **Changing data**
   It is a known fact that if the data incoming to the processor ALUs changes over time, the dynamic power consumption of the ALUs is higher. However, we don't know whether this causes a relevant, measurable change in temperature.

   We run benchmarks for integer addition (both 32 and 64-bit), with two types of input data – static and changing. The static benchmark adds 0-s the entire time; the changing benchmark operates with 4 registers in 2 stages:

a. $r_2 = r_0 + r_1 \equiv -1 = 0 + -1 \equiv \texttt{FFFFFFFF} = \texttt{00000000} + \texttt{FFFFFFFF}$
b. $r_0 = r_2 + r_3 \equiv \quad 0 = -1 + 1 \equiv \texttt{00000000} = \texttt{FFFFFFFF} + \texttt{00000001}$

The Least Significant Bit of the 2nd input doesn't change, so this is not a perfect solution, but the value of all other input/output bits changes on each step, so it should work sufficiently well for our measurement.

All experiments were conducted on the NVIDIA TX2 platform, on the four ARM Cortex-A57 cores. The CPU was cooled down to a standard temperature (usually 40 °C) between benchmark runs to be able to make meaningful comparisons.

3.4.2 Results

I visualized and interpreted the results using the Thermobench report generator tool described in section 3.2. The graphs illustrating the experiments are mostly slices of the originals in order to be better readable even when they are smaller.

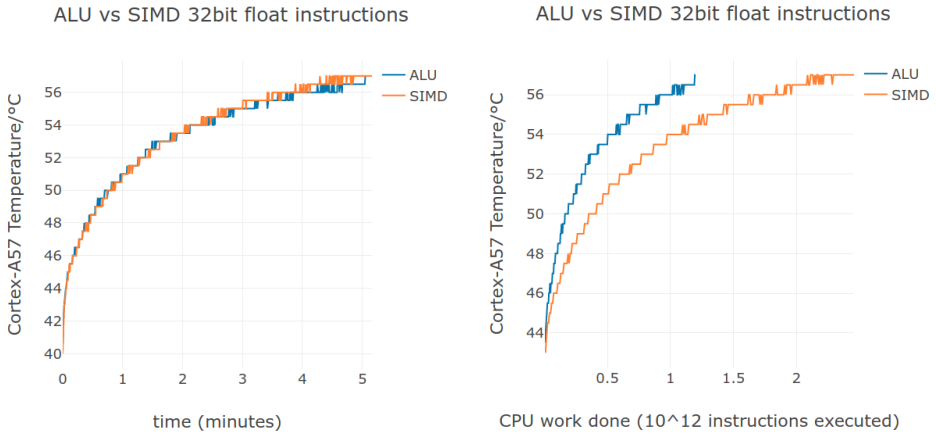
1. **SIMD vs ALU execution**

   For most instructions there is very little difference in terms of temperature between ALU and SIMD instructions (see Figure 3.4a). Given that the SIMD module does multiple instructions worth of work on a single cycle, it is well worth to use it (see Figure 3.4b), as more work can be done for the same temperature increase.

2. **Arithmetic instruction comparison – ADD, MUL, MADD, DIV**

   Float 32 & 64 bit
   The ADD, MUL, MADD instructions are roughly thermally equivalent; DIV is cheaper than any of them – this is consistent through both ALU and SIMD, 32 and 64bit instructions. To keep the graph legible, Figures 3.5a and 3.5b show 4 ALU 32bit float instructions.

   ALU integer 32 & 64 bit
   For ALU integer instructions, ADD instructions are the most expensive; after that DIV instructions, and MUL and MADD instructions cause roughly the same, smallest temperature increase (Figure 3.5d). DIV heats up the CPU less than any other instruction (Figure 3.5c), however, it also works slower, and thus has a worse ratio of performance and



(a) : Temperature change during run w.r.t. time

(b) : Temperature change during run w.r.t. CPU work done (total number of additions)

Figure 3.4: ALU vs SIMD instructions 32bit float addition

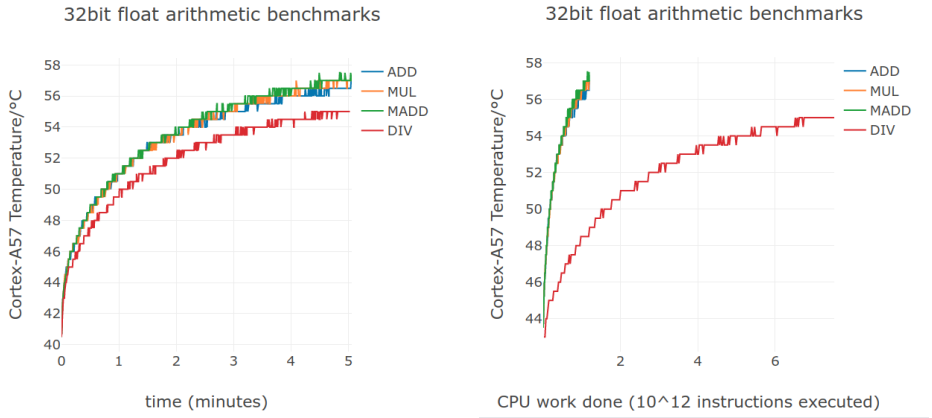
temperature increase than MUL and MADD instructions. The figures show only ALU int32 instructions, again to keep the graphs legible.

SIMD integer 8/16/32/64 bit
For integer SIMD instructions, ADD causes smaller temperature increase than MUL & MADD. There is no instruction for SIMD division in the ARM NEON module.

3. **Multiply + Add vs MADD**

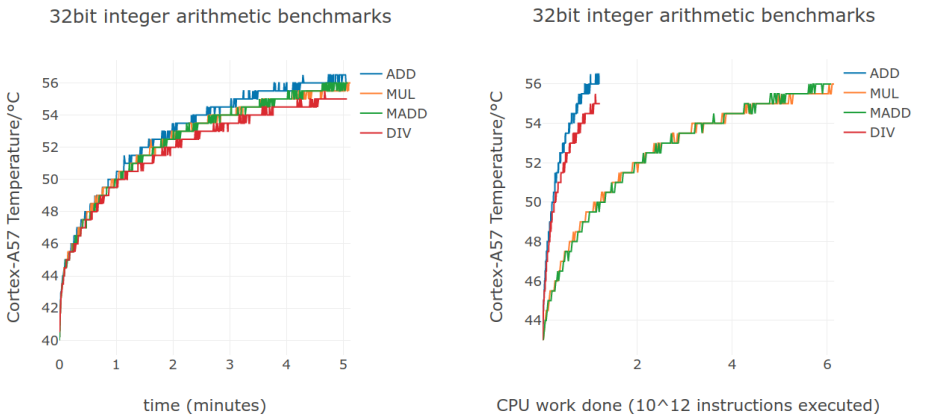
As seen on Figures 3.6a and 3.6b, for ALU instructions MADD heats the CPU up in a shorter time, and is less thermally efficient. For SIMD instructions, MADD instructions are more efficient, although only slighly (Figures 3.6c and 3.6d).

4. **Changing data**
Comparing Figures 3.7a and 3.7b we can see that the effect of the changing data is only relevant for SIMD instructions, for ALU instructions it is negligible.



(a) : 32bit float Time-Temperature

(b) : 32bit float Work-Temperature



(c) : 32bit float Time-Temperature

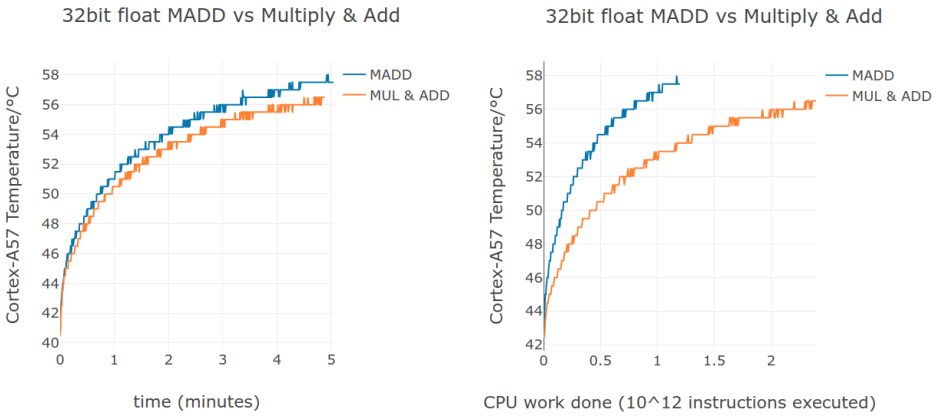
(d) : 32bit float Work-Temperature

Figure 3.5: Arithmetic benchmark examples

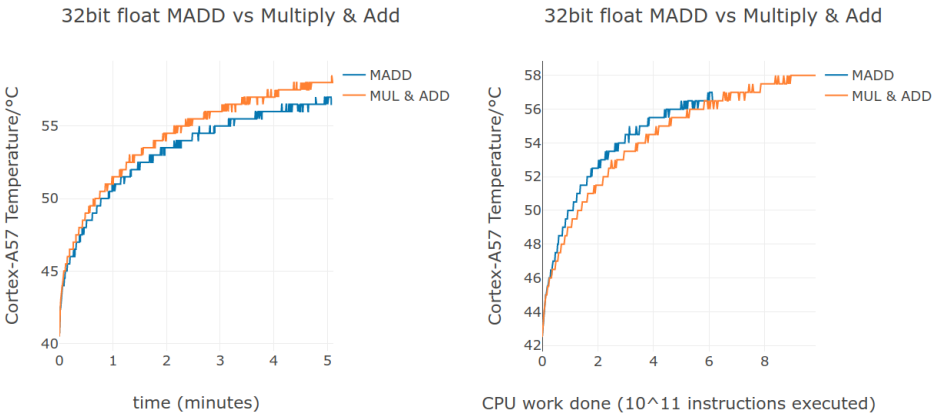
3.4.3 Conclusion

To summarize the results of our experiments:

1. SIMD instructions cause the same temperature change over time as ALU instructions, but do more work per cycle, so they should be used whenever possible
2. For applications with a lot of divisions, it is advisable to use floats over integers. ALU integer multiplications are generally cheaper than additions – for integer SIMD instructions the reverse is true.
3. Only SIMD MADD instructions are more thermally efficient than MUL+ADD, for ALU instructions the reverse is true
4. Data variance only has a measurable effect when using SIMD instructions

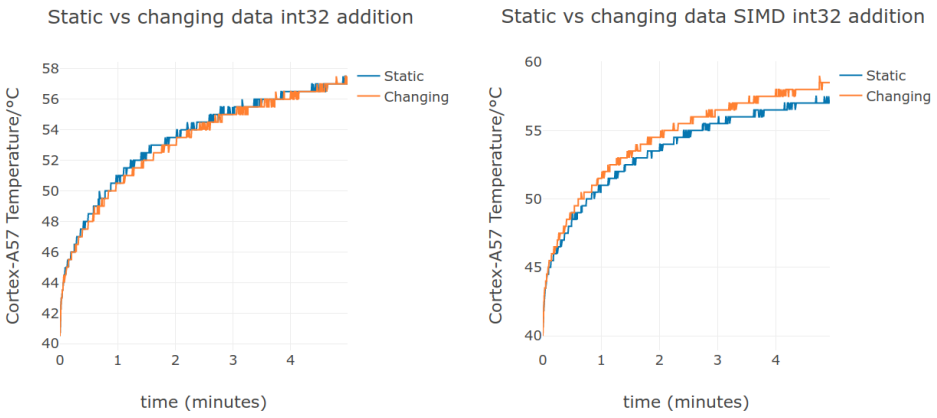


(a) : Time-Temperature **(b) :** Work-Temperature



(c) : SIMD Time-Temperature **(d) :** SIMD Work-Temperature

Figure 3.6: 32bit float MADD vs Multiply & Add



(a) : ALU Time-Temperature **(b) :** SIMD Time-Temperature

Figure 3.7: The effect of changing data on int32 addition

Chapter 4

Tracking in image processing

As mentioned in chapter 1, this chapter provides a basic background in point and object tracking, which is required for the comprehension of the analysis steps in chapter 5 describing the Thermocam-PCB tool.

4.1 Tracking fundamentals

Images are perspective projections of 3D points to an image plane (see Figure 4.1). The projection is one-to-one when all objects are completely nontransparent – we will ignore the problem of tracking transparent objects in this chapter.

Any algorithm for tracking a set of points on nontransparent objects through a time series of images is defined by the following two steps:

1. **Initialization** – Select a set of 2D points in the initial image. This set has a corresponding 3D point set, let us call it S^3. Note that S^3 can only consist of 3D points facing the camera in the initial image, as we constrained the perspective projections to be one-to-one.
2. **Tracking step** – In all other images: find the set of 2D points that corresponds to the 3D point set S^3.

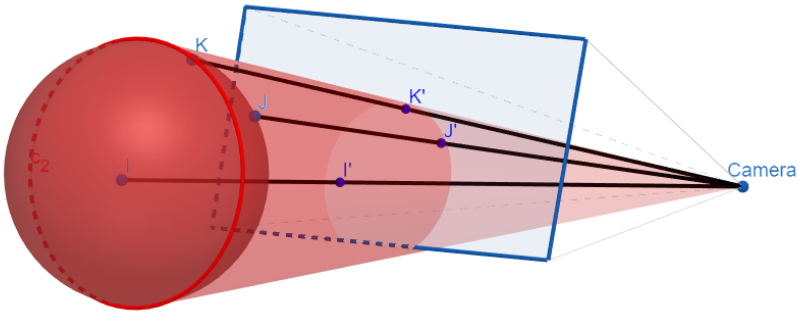


Figure 4.1: Projection of a side of a sphere to the image plane
3D points {I, J, K} project to 2D points {I', J', K'} in the image plane

For some applications, the previous definition is insufficient – we may want to track all visible points belonging to a chosen object, not just the ones initially facing the camera (e.g. when tracking a rotating ball – see Figure 4.1). Although this seems a minor detail, it has large implications on the structure of the algorithms, discussed later in this section.

Tracking algorithms have to cope with the following changing factors:

1. Camera
   - 1.1. Position
   - 1.2. Rotation
2. Object
   - 2.1. Position
   - 2.2. Rotation
   - 2.3. (Self)Occlusion
   - 2.4. Color
   - 2.5. Structure (people moving, etc.)
3. Scene Illumination

All tracking algorithms (through the tracking step defined at the beginning of the section) are designed to be robust to 1.1, 1.2, 2.1 as long as occlusion is minimal. Many algorithms are comfortable with this default constraint, and are used in scenarios with almost no occlusion.

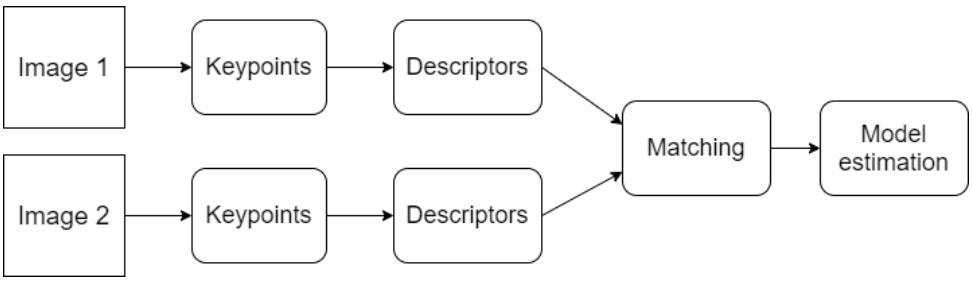


Figure 4.2: Stages of the correspondence matching pipeline

Algorithms that want to track objects through appearance change and occlusion need to iteratively update their model of the object, as they usually do not have a concept of 3D space. However, often this results in the model drifting away from the object, and the tracker tracking an arbitrary part of the scene, without the possibility of recovery.

Another approach is to assume a static scene, or a scene dominated by a rigid object. This way we do not need to solve the appearance change problem and we can find the projective transformations between the two images based on 3D perspective geometry. The process to find the transformation is detailed in the next section.

4.2 Correspondence problem and solutions

The goal of correspondence matching is finding points in 2 images that are projections of the same 3D point in space. With this information we can estimate the perspective transformation matrix between the two images. Correspondence matching has the form of a pipeline (see Figure 4.2), its main stages are:

1. **Keypoint** detection – detect projections of 3D points that can be easily recognized and localized in images from different 3D camera positions
2. **Descriptor** construction – create a feature vector describing the keypoint neighborhood
3. Descriptor **matching** – create valid pairs of keypoints between images
4. Robust **model** estimation – use pairs to estimate perspective transformation matrix

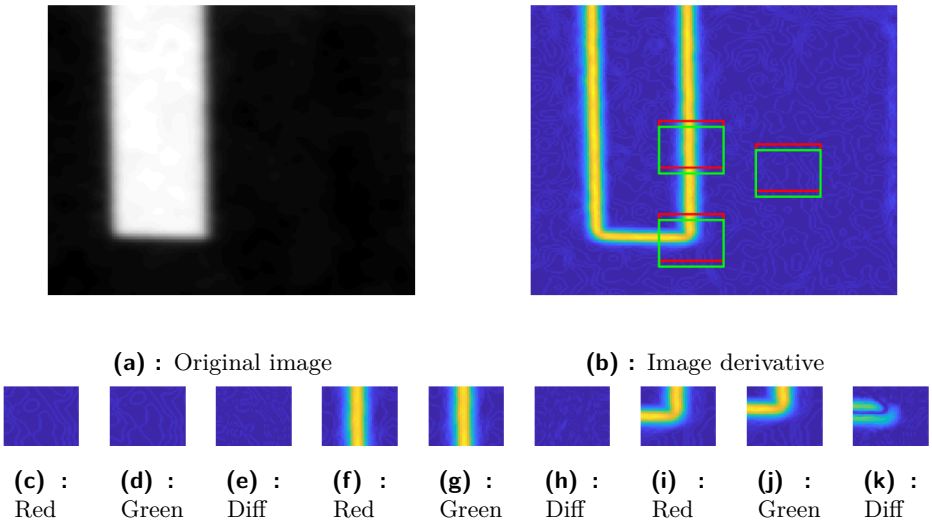
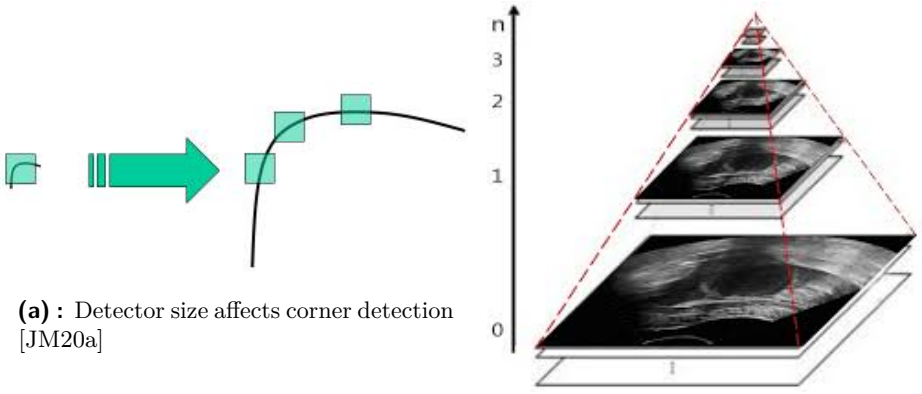


Figure 4.3: Point neighborhoods (red rectangle) moved slightly off-center (green rectangle), and their difference. Corners (4.3k), differ from their surroundings in all directions, as opposed to homogeneous surfaces(4.3e) or edges (4.3h).

4.2.1 Keypoint detection

Corner detection forms the base of almost all keypoint detectors. Corners are easily distinguishable from their surroundings – movement in any direction causes significant change in the point neighborhood (see Figure 4.3). This makes them great candidates for precise matching.

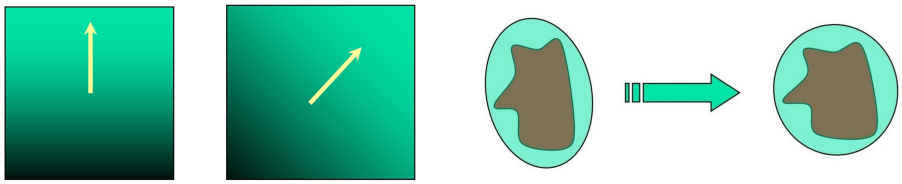
Multi-scale detection is often also useful – on sufficiently small scales every curve is a straight line, thus there are no corners to detect (see Figure 4.4a). The scale layers (Figure 4.4b) are called **octaves** in the literature.



(a) : Detector size affects corner detection [JM20a]

(b) : Scale-space octaves [LYZ+14]

Figure 4.4: Multi-scale detection



(a) : Normalization by dominant gradient orientation [JM20b]

(b) : Affine normalization by geometric moments [JM20c]

Figure 4.5: Keypoint normalization methods

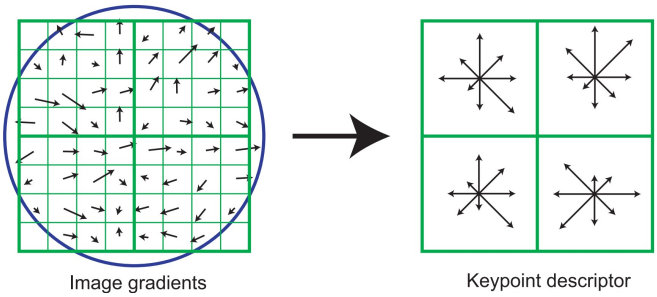
4.2.2 Descriptor construction

The goal of a descriptor is to represent a point's local neighborhood in a compact but informative way. **Normalization** of that neighborhood improves descriptor robustness, so it often precedes construction. The final descriptors can be **SIFT-like** or **Binary** (see [Ort12]), depending on the method used.

Normalization

To make descriptors of keypoints corresponding to the same 3D points more similar to each other, keypoint neighborhoods from which the descriptors are constructed are often normalized with respect to:

- **Scale** – Select image patch from octave stored during keypoint detection
- **Rotation**
  1. Determine dominant gradient orientation (Figure 4.5a)
  2. Rotate the image patch to 0 rotation
- **Affine transformations**
  1. Approximate neighborhood with ellipse through covariance matrix
  2. Compute geometric moments of orders up to 2
  3. Normalize affine region (ellipse) to circular one (Figure 4.5b)



(a) : 4 HOGs from 4 sections, 4×16 subsections (SIFT) [Low04]

(b) : DAISY [TLF10] sampling pattern, 1 HOG per circle.

Figure 4.6: Descriptor sampling patterns. Gradients are weighed with a 2D Gaussian with variance dependent on circle size.

SIFT-like descriptors

Scale Invariant Feature Transform (**SIFT**) [Low99] is an exceedingly influential, industry standard keypoint detection and descriptor construction method. **SIFT-like** methods are either improvements on the original SIFT, or use very similar principles of operation.

Histograms of **Oriented Gradients** (**HOGs**) [DT05] are the basic building blocks of SIFT-like descriptors. They are calculated as follows:

1. Split the keypoint neighborhood into **sections** and those into **subsections**.
2. Calculate the **dominant gradient orientation** for each subsection.
3. (Optional) **Weigh gradients** based on distance from the center of sampling. This reduces noise from the point neighborhood border.
4. **Sample** and **group** the orientations **into histograms** for each section.

The final descriptor is a vector containing all histograms around the keypoint. Its robustness can be further improved by additional thresholding, normalization etc.

(a) : BRISK [LCS11] **(b) :** FREAK [Ort12]

Figure 4.7: Sampling patterns for binary descriptors

Binary descriptors

Binary descriptors are **vectors of binary features** (often 1 bit, 2–3 for more complex features). The features are assumed to be independent and of equal importance – this allows descriptors to be compared using Hamming distance.

To generate these features, binary descriptor methods **sample image intensities** at specific places (see Figure 4.7) in the point neighborhood and **compare** them according to their method-specific rulesets.

4.2.3 Descriptor matching

Descriptor matching aims to pair together descriptors which correspond to the same 3D point in space. It is based on **distance** – **Hamming** for binary and **Euclidean** for SIFT-like descriptors. It can employ strategies such as:

1. **Mutually nearest** – match each descriptor to its nearest neighbor
2. **Stable pairing** – all descriptors can only be matched once:
   - 2.1. Rank descriptors by ascending distance
   - 2.2. Match closest pair
   - 2.3. Exclude all other pairs containing descriptors from the pair just matched
3. **First/second nearest** – Mutually nearest + the distance ratio of the first/second nearest neighbors must be above threshold (≈ 0.8 [Low04])

Matching is generally the most **computationally demanding** part of the pipeline. Thus, in real-time applications it is often worthwhile to use

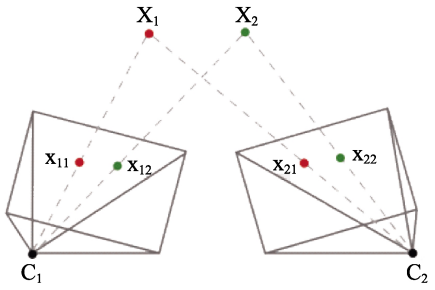


Figure 4.8: Image pairs with corresponding points [LYC+19]

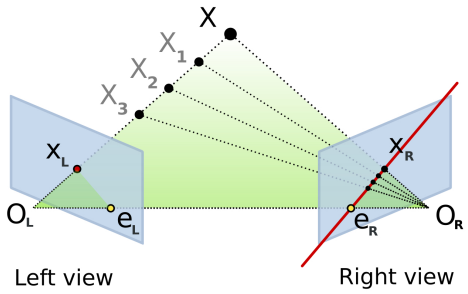
approximate nearest neighbor algorithms instead of all-to-all comparison to trade off some accuracy for speed.

4.2.4 Robust model estimation

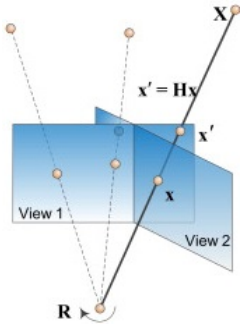
The result of correspondence matching is a set of pairs of 2D points, each pair corresponding to a 3D point in space (see Figure 4.8). We can use these pairs of 2D points to approximate the geometric transformation between the 2 image planes. The transformation model is a 3x3 matrix that transforms any 2D point in one image to a corresponding 2D point in the second image:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = M \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \tag{4.1}$$

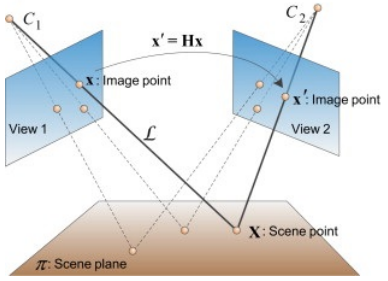
There are 3 main types of perspective transformation models to consider in this case: **Fundamental matrix**, **Homography** and **Affine transformation matrix**. For each of the models, the number of point pairs required to construct them is equal to their degrees of freedom.



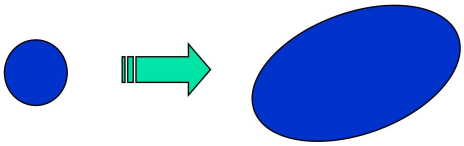
(a) : Epipolar geometry of 2 image planes, which defines the fundamental matrix[Wik20]



(b) : Homography – rotation, no translation [KWW+14]



(c) : Homography – rotation and translation above a plane [KWW+14]



(d) : Affine transformation [JM20d]

Figure 4.9: Perspective transformation model types

Model types

1. **Fundamental matrix** – uses epipolar geometry to describe the transformation between the perspective camera projection matrices of the 2 images (Figure 4.9a). It is usable for arbitrary scenes, and requires **7 pairs of points** for construction.
2. **Homography** – special case of fundamental matrix, is equivalent to the fundamental matrix in the following cases:
   a. Rotation of the camera, no translation (Figure 4.9b)
   b. Translation + rotation of the camera above a plane (Figure 4.9c)

   It is often used as a simplification of the fundamental matrix (even for scenes not precisely described by the 2 cases above), as it only requires **4 pairs of points** for construction.
3. **Affine transformation matrix** – describes 2D translation, rotation and non-uniform scaling (Figure 4.9d). Used when speed is much more important than precision, requires **3 pairs of points** for construction.

Method	Main idea
LMEDS	Minimize median error
RANSAC	Maximize number of inliers
PROSAC	RANSAC + sampling probability depends on match distance
LO-RANSAC	RANSAC + local optimization on model update
MSAC	RANSAC + quadratic threshold
MLESAC	RANSAC + maximum likelihood threshold

Table 4.1: Examples of common robust model estimation methods

Robust estimation

Most keypoint detection/description methods produce a significant number of wrong correspondence pairs – outliers. Thus, only methods which can deal with a large number of outliers (robust methods) are practically feasible to use for model estimation. Variants of the **Random Sample Consensus** (**RANSAC**) method and the **Least Median of Squares** (**LMEDS**) method are common choices. Their basic structure is the following:

Input: Set of correspondence pairs
Output: Perspective transformation matrix

Method steps:

1. Sample points from dataset, create a model from them.
2. Check if the current model is better than the best one so far. The criterion function depends on the method used:
   - LMEDS: Minimal median of squared residual reprojection error
   - RANSAC: Most points within reprojection error threshold (hyperparameter)
3. If the current model is better, update the best model
4. If number of maximum iterations (hyperparameter) is reached, stop
5. Repeat from step 1

Table 4.1 shows some methods commonly used for model estimation. Which method is used is usually determined empirically, for each application.

	Inliers					
Sampled points	15%	20%	30%	40%	50%	70%
2	132	73	32	17	10	4
4	5916	1871	368	116	46	11
7	1.75×10^6	2.34×10^5	1.37×10^4	1827	382	35
8	1.17×10^7	1.17×10^6	4.57×10^4	4570	765	50
12	2.31×10^{10}	7.31×10^8	5.64×10^6	1.79×10^5	1.23×10^4	215

Table 4.2: Mean RANSAC iterations required for an all-inlier sample [JM20e]

The choice of model type is a very important aspect for any method using random sampling, both in terms of performance and precision. Table 4.2 shows that as the ratio of inliers decreases, it becomes less likely to get an all-inlier sample. Thus, many applications cannot afford to use models with a high number of required points (e.g. 7 for the fundamental matrix).

(a) : Naiive location filter (1 at object position, 0 everywhere else)

(b) : Gaussian location filter (naiive filter convolved with Gaussian)

Figure 4.10: Correlation filters

4.3 Object tracking methods

This section details object tracking methods which were both considered as a tracking solution in chapter 5, and selected as workloads to test temperature reduction methods on in chapter 7.

4.3.1 Kernelized Correlation Filter (KCF) Tracker

KCF [HCMB14] is an example of correlation-based tracking, which aims to find a **correlation filter** that, after applying it to the input returns **1 at the position of the object**, and **0 everywhere else** (Figure 4.10a). Requirement for such a sharp peak often leads to **overfitting** – some strategies to reduce it are:

1. **Convolve** the reference peak **with** a **Gaussian** – smoother learning (Figure 4.10b). The optimization problem for a Gaussian peak is:

$$\min_w \|x \circledast w - g\| \tag{4.2}$$

x – input image, $\circledast$ – convolution operator, w – correlation filter, g – reference location spike convolved with a Gaussian.

2. **Circular matrix** as emulation of a larger dataset (Figure 4.11):

$$C(x) = \begin{bmatrix} (P^0 x)^T \\ (P^1 x)^T \\ \dots \\ (P^{n-1} x)^T \end{bmatrix} \tag{4.3}$$

Here, P is a permutation matrix shifting the image by 1 pixel in the vertical/horizontal direction. With this the optimization problem becomes:

$$\min_w \|C(x)w - g\| \tag{4.4}$$

Figure 4.11: Circular matrix of image preprocessed with cosine window [JM16]

3. **Ridge regression** formulation

$$\min_w \|C(x)w - g\| + \lambda \|w\|^2 \tag{4.5}$$

Kernel ridge regression

The dual form of the ridge regression formulation allows us to replace the dot product with a **nonlinear kernel**, massively increasing the feature space without compromising on performance:

$$\min_w \|K\alpha - g\| + \lambda \alpha^T K \alpha \quad \Longrightarrow \quad \alpha = (K + \lambda I)^{-1} g \tag{4.6}$$

Solving in the Fourier domain (given that $K = C(k)$) yields:

$$\hat{\alpha} = \frac{\hat{g}}{\hat{k} + \lambda} \tag{4.7}$$

Model update

KCF is not robust to object rotation or scale change. Thus, it needs to update its model through **linear interpolation**:

$$\alpha_k = (1 - \eta)\alpha_{k-1} + \eta \alpha_{new} \tag{4.8}$$

The more the appearance of the object changes, the larger η needs to be. However, increasing it also increases tracker drift.

Summary

The main strength of KCF trackers is their speed – given the computational effectivity of convolutions and circular matrices in the Fourier realm, they often are easily real-time. Their main weakness comes from the fact that after almost any 2D transformation of the object projection (2D rotation, affine transform, etc.) the object model needs to be modified as if it had completely changed its appearance.

4.3.2 Kanade-Lukas-Tomasi (KLT) Tracker

For the KLT tracker [TK91], the main steps of tracking defined in the beginning of chapter 4 are:

1. **Initialization**: Find keypoints inside object bounding box. The neighborhoods of these keypoints will be called **template patches**.
2. **Tracking step**: For each image and keypoint iterate:
   a. Calculate the difference of the template and current patch. The initial current patch is the neighborhood of the point from the current image with the coordinates of the keypoint from the previous image.
   b. Compute gradient of difference, estimate displacement
   c. Move towards gradient, update coordinates of current patch
   d. Repeat from a. until convergence

The tracked points can be used to calculate a perspective transformation model for the movement of the object. As the number of tracked points is usually low compared to pixels in the image, this method is an example of sparse optical flow – see [SES10].

The KLT tracker sits in the category of object tracking algorithms that only track points initially facing the camera and are not robust to object appearance change (see section 4.1). It is thus robust to non-occluding object rotations and scale change. However, it is not suitable for long term tracking even for scenes with little occlusion – once a keypoint is lost, it is lost forever.

Chapter 5

Thermocam-PCB tool

The main purpose of the Thermocam-PCB tool is to read, process and publish the image stream data from the **Workswell Infrared Camera (WIC)** – see section 2.2. Using an infrared camera complements the Thermobench tool (chapter 3) by allowing us to:

1. Measure the temperature of chips external to the main SoC
2. Validate our measurements made with Thermobench

The tool is designed to measure the temperature on a **Printed Circuit Board (PCB)**. Most of its features can be used for any scene, but the tracking feature is limited to 2D static scenes, or mostly planar, rigid objects which dominate the scene and have a homogeneous (for the infrared camera) background behind them. A PCB lying on a room-temperature table is an example of such a scene. This limitation enables the tracker to use simpler models of the scene and achieve better performance and precision because of that.

In the THERMAC project, the tool is used to monitor the i.MX8 platform, and runs on the Minnowboard Turbot platform – details in chapter 2.

5.1 Requirements

The Thermocam-PCB tool needs to satisfy the following functional and quality requirements:

5.1.1 Functional requirements

- Display the WIC image stream
- Enable the user to enter Points Of Interest (POIs) anywhere on the image and display their temperature
- Save and Import POIs and the current WIC image into JSON file
- Track POIs across the image stream
- Publish WIC image stream and POIs through a webserver
- Record video
- Set recorded video as input

5.1.2 Quality requirements

1. Precise temperature measurement ($\pm 0.5°C$)
2. Precise point tracking (max 2-3px error)
3. Real-time computation on an embedded platform (<400 ms)
4. Detectable & recoverable failure modes
5. Robustness to local & global change of temperature over surfaces with variable emissivity

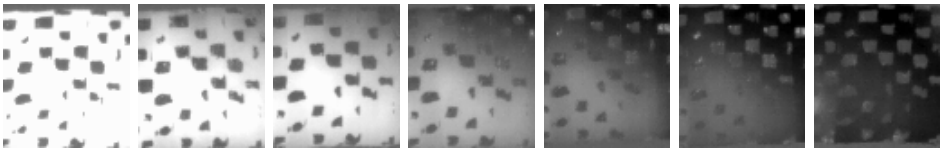


Figure 5.1: Cooldown of checkered pattern made of low thermal emissivity (≈ 0.03) aluminium foil and high thermal emissivity (≈ 0.95) electrical insulating tape

5.2 Related work

Most functional requirements of the tool (e.g. displaying the image stream through a webserver or recording video) have straightforward solutions and implementations. The only functionality that required research into scientific literature is point tracking through the thermal image stream.

Most relevant publications solving the problem of tracking in thermal imaging use image registration, usually through the correspondence matching pipeline (see section 4.2) to rectify the images either to a top-down or the initial viewpoint – e.g. [and17],[RLSB19],[SC19],[DC19],[CLmS16], [KLB+17]. Object tracking algorithms such as KCF and KLT (see section 4.3) are used more rarely – [CH14], [AN17]. Some publications use FFT phase correlation to rectify small movements, such as vibrations – [HGMLHM12],[Mod11].

The publications mentioned above do not solve the problem of the temperature of the tracked object changing over time, except for [CH14], which circumvents the problem by masking the parts of the image where the temperature changes significantly. [and17] uses Contrast Limited Adaptive Histogram Equalization that may help in the matter but it is used more for improving contrast rather than counteracting temperature change.

Unfortunately, to normalize thermal images through temperature change we cannot use algorithms used to compensate for lighting changes in visible-light images. Surfaces with low thermal emissivity have more or less constant intensity through temperature change compared to high emissivity surfaces. This means that the contrast between two surfaces may disappear and even switch around – see Figure 5.1. Publications that attempt the registration of thermal and visible-light images might help in this problem: [JPR+07] proposes texture filters to normalize both types of images to have a unified visual appearance.

5.3 Design & Analysis

As discussed in the previous section, point tracking was the only feature of the tool that required research into the literature. It was also the only feature that required thorough analysis. The previous section established that most publications in the relevant literature use correspondence matching based image registration or object tracking methods, and for most of them the temperature of the tracked objects does not change significantly.

When running benchmark applications testing the thermal properties of the chips on our hardware platforms (see section 2.1), the temperature of the scene changes globally (the entire PCB heats up) and also locally (the SoC heats up more than its surroundings). There are two major ways to solve this problem:

1. **Use a tracking algorithm robust to global and local temperature change**
   Such tracking algorithms would require some way to ensure robustness to object appearance change. As discussed in the previous section, methods robust to lighting change cannot be successfully used to compensate for changes in thermal images. Thus, we need robustness to any change in object appearance.

   Such algorithms exist, however, as discussed in section 4.1, they introduce drift, which can lead to unrecoverable failure. As one of the quality requirements established in section 5.1 is the ability to recover from a failure state, such methods are not usable.

2. **Preprocess the image for the algorithms that work on non-changing temperatures**
   As previously mentioned, there is an array of solutions that work in scenes where the temperature of objects does not change significantly. If we can find some method of preprocessing the images such that they would be normalized for global and local temperature changes sufficiently for the algorithms to be able to use them, we could satisfy all the requirements for the tool.

Only the second option could viably keep all requirements from section 5.1, so that was the one we chose.

5.3.1 Choosing the preprocessing method

The ideal preprocessing method would normalize the image for temperature change such that images from the same viewpoint would be indistinguishable with any change in object temperature, while preserving as much visual information as possible for the tracking algorithms to use.

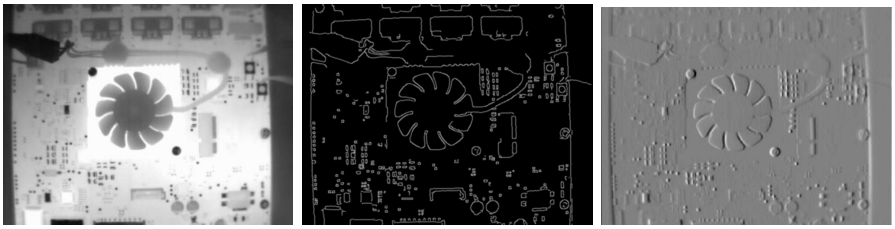
In reality we are unlikely to find such a preprocessing method, however, we can compare them according to how much detail they remove and how well they eliminate global and local temperature changes. We tried a few methods, most of which were used in the relevant literature:

- **Canny edge detection** (Figure 5.2b)
  Images from the same viewpoint are similar, but removes way too much detail.
- **Texture filters** (Figure 5.2c)
  Used in [JPR+07], as a preprocessing step for the registration of color and thermal images. Preserves the similarity across temperatures, but is low on details.
- **Histogram equalization** (Figure 5.2d)
  Used in [DC19], compensates global temperature changes well, amplifies local ones. Retains details well.
- **Contrast Limited Adaptive Histogram Equalization** (**CLAHE**) (Figure 5.2e)
  Used in [and17]. Local normalization, compensates for both global and local temperature change well. Preserves details, but tends to amplify input noise.

We chose **CLAHE**, as it preserved the most details while providing good local normalization. As CLAHE amplifies noise, we apply **median filtering** before using it on the image, and then use **unsharp masking** to reduce the blur introduced by the median filter. The result can be seen on Figure 5.2f.

5.3.2 Choosing the tracking algorithm

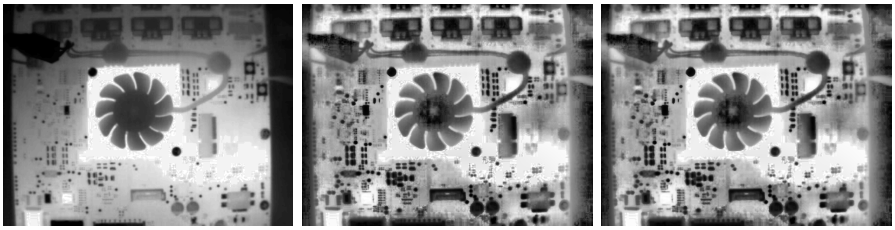
Section 5.2 mentions several tracking algorithms that have been successfully used on thermal image streams. The list below details their limitations:



(a) : Original image

(b) : Canny edge detection

(c) : Texture filter



(d) : Histogram equalization

(e) : CLAHE

(f) : Median filter, CLAHE, Unsharp masking

Figure 5.2: Preprocessing techniques

1. **Object tracking algorithms**

   - KCF - Template update required for scale change and rotations, which introduces drift
   - KLT - Loses points over time, leading to unrecoverable failure

2. **Correspondence matching & perspective transform estimation**

   - Requires the scene to be rigid (true when most of the scene is occupied by the PCB)
   - Consecutive images are treated as independent
     - Immune to drift & implicit recovery from failure
     - Discards a large amount of information (successive frames are very similar in practice)

3. **FFT phase correlation**
   Can only be used for very small movements, such vibrations.

We chose **correspondence matching**, as it satisfies the requirements better than the presented alternatives. It can recover from failure states, is fairly precise and can be implemented to run in real-time. Its main drawback is that it doesn't utilize the fact that successive frames are similar.

Choosing the perspective transformation model for correspondence matching

After we chose correspondence matching as our method of tracking, we needed to decide what kind of perspective transformation we need to approximate in our application. We established 3 transformation model types in section 4.2.4:

1. Fundamental matrix
2. Homography
3. Affine transformation matrix

Given that the background behind the PCB is featureless for the thermal camera and all points lie on the PCB, which is a rigid, mostly planar object, any movement of the PCB on the table can be described by an equivalent camera transformation. We can thus describe the setup roughly as the movement of a camera over a plane, which is one of the special cases where the fundamental matrix is equal to a homography.

Using a homography also allows much faster computation (see 4.2.4), because it requires only 4 points to construct. We did not use the affine transformation model, because translation, rotation, scale and shear simply does not cover all types of transformations relevant for our use case.

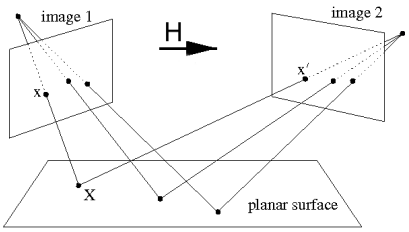


Figure 5.3: Camera movement over plane, described by homography [HZ04]

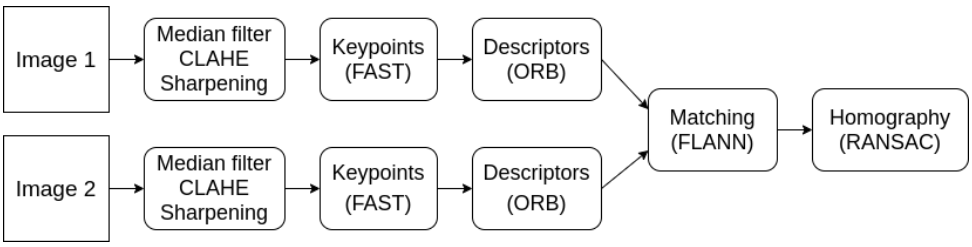


Figure 5.4: Initial correspondence matching pipeline

5.3.3 Tuning the chosen tracking algorithm

There are many variants of the correspondence matching pipeline (described in section 4.2), multiple methods can be used for all of the pipeline stages, each with their own hyperparameters. Both the choice of the method and hyperparameters for the given method highly influence how well the pipeline performs.

Initially, we arbitrarily chose some popular methods for the pipeline stages with their default hyperparameters (see Figure 5.4). Except for the simplest transformations (identity, small translations), the results were inaccurate and changed significantly (± 10 px) from frame to frame. Thus, it seemed worthwhile to search for the best possible combination of methods and hyperparameters in the pipeline for our specific application.

We considered using some of the many method comparison studies (e.g. [JC16], [BB17]) to at least select the best method combination for the pipeline. However, these studies use visible-light images with a variety of scenes for testing, while we needed optimal parameters for a thermal image stream of a PCB – it is unlikely that the optimal pipeline is the same for those two scenarios.

Table 5.1 shows all methods and hyperparameters that we searched for all pipeline stages. We first created a dataset on which we could test the pipeline variants (detailed in the next section). Then, for each pipeline stage, we froze the parameters for all other stages and searched for the best hyperparameters for that stage. The only exception in this were the keypoint detection and descriptor extraction stages, which we optimized together, trying out all possible method combinations.

By optimizing the pipeline stages separately it was likely that we would only get to a local optimum of the method and hyperparameter combinations. However, it was computationally infeasible to optimize all stages at the same time, and we could still significantly improve tracking precision, as shown by the results of the hyperparameter search at the end of this section.

Pipeline stage	Method	Hyperparameter
Preprocessing	Median filtering	Kernel size
	CLAHE	Threshold
		Kernel size
	Unsharp masking	Kernel size
Keypoint detector	FAST	Detection threshold
		Non-max Suppression
		Detector Type
	BRISK	Threshold
		Octaves
	ORB	FAST Threshold
		Max Features
		Octaves
		Octave scale factor
		Detector size
	MSER	Delta
		Minimum area
		Maximum area
	AGAST	Threshold
		Non-max Suppression
		Detector Type
	AKAZE	Threshold
		Octaves
		Layers per octave
		Diffusivity type
Descriptor extractor	BRISK	Patch size
	FREAK	Orientation normalization
		Scale normalization
		Pattern scale
		Octaves
	ORB	Patch size
	LATCH	Descriptor bytes
		Rotation invariance
		Mini-patch size
		Gaussian blur sigma
	AKAZE	$\emptyset$
Matching	First/Second threshold	
Model estimation	RANSAC	Threshold
		Confidence interval
		Maximum iterations
	PROSAC	Threshold
		Confidence interval
		Maximum iterations
	LMEDS	$\emptyset$

Table 5.1: All pipeline stages, methods and hyperparameters searched

Parameter	Max/Min	Meaning
t_x, t_y	$\pm 0.25\times$ image width,height	Translation in x,y direction
ϕ	$\pm\frac{\pi}{2}$	Rotation angle around center
s_x, s_y	1.2/0.8	Scale in x,y direction
sh_x, sh_y	± 0.1	Shear in x,y direction
p_1, p_2	± 0.0003	Projective parameters

Table 5.2: Homography parameter ranges and descriptions

Dataset

The input dataset for the correspondence matching pipeline consists of pairs of images that simulate camera movement over the PCB – one is a reference, the other is transformed by a homography. The homographies are constructed from Euclidean, Scale/Shear and Projective transformation matrices (see [Thi17]):

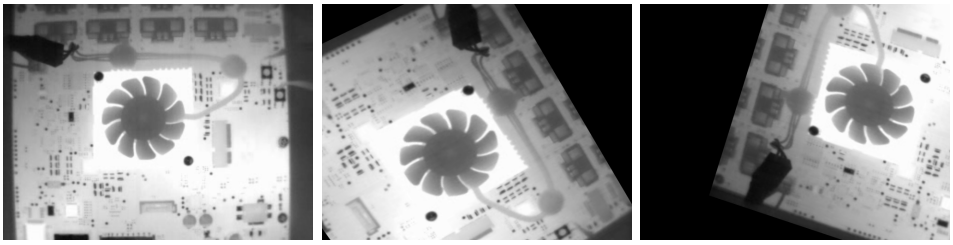
$$H = ESP = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & t_x \\ \sin(\phi) & \cos(\phi) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & sh_y & 0 \\ sh_x & s_y & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ p_1 & p_2 & 1 \end{bmatrix} \tag{5.1}$$

The meaning of the matrix parameters, as well as their min/max values are detailed in Table 5.2. We set the min/max parameters by hand, by looking at the images transformed by the homographies and choosing ones that reflect transformations that we would realistically see when moving the camera around.

To generate a random homography for the dataset, the program samples parameters from a uniform distribution with min/max values detailed in Table 5.2 and constructs the matrix according to equation 5.1.

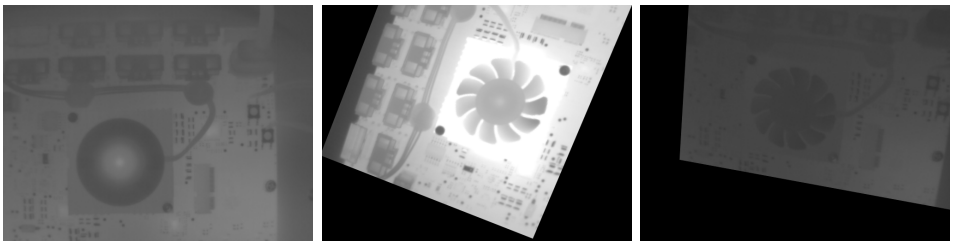
The main advantages of generating the set of image pairs this way are:

- Closely controllable complexity – changing homography parameters allows setting exactly what kind of projections we want
- Arbitrary size – many possible pairs from the image set
- Easy result checking – we have a reference to compare pipeline results to



(a) : Original **(b) :** Transformed 1 **(c) :** Transformed 2

Figure 5.5: One original and two transformed images from the small image set



(a) : Unchanged **(b) :** Transformed 1 **(c) :** Transformed 2

Figure 5.6: One unchanged and two transformed images from the large image set

We created two datasets of image pairs, one smaller for rapid testing, and a second larger one for more thorough, temperature-invariant testing:

- **Small dataset**
  - **Input:** 5 reference images of the board from different angles and with different temperatures
  - **Output:** Each of these 5 images transformed by 3 random homographies each – 15 image pairs (see Figure 5.5)
- **Large dataset**
  - **Input:** 220 images of the board, from the same angle, with different temperatures – sampled through a long time period with workloads periodically applied to the SoC, shuffled afterwards
  - **Output:** First 20 images unchanged, the other 200 transformed by random homographies – 200 image pairs, 10 transformed for each unchanged image (see Figure 5.6)
  - Any pair of unchanged-transformed images can be of very different temperatures – the optimal pipeline variant has to be capable of handling temperature change well

Metric of comparison

To compare two pipeline variants, we need a metric by which we can compare them. The requirements we set for the metric are the following:

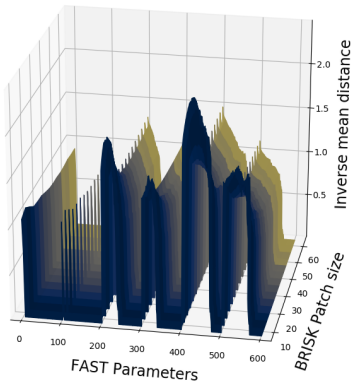
- **Measure the quality of the pipeline variant** – how close is the homography found to the reference homography.
- **Higher score = better variant** – it is visually more convenient to look for maxima on the graph than minima
- **Capped minimum** – improves visualization, low and extremely low scores are equally unusable for us
- **Independent of the model estimation stage**.
  In the initial pipeline, I used the RANSAC algorithm for the model estimation stage. Threshold, its most important parameter heavily depends on the number and quality of correspondence pairs found, which in turn heavily depends on the keypoint and descriptor method parameters. Thus I either needed to find the optimal RANSAC threshold for each parameter combination (computationally infeasible), or find a metric that does not use RANSAC or model estimation at all.

The metrics I tried were:

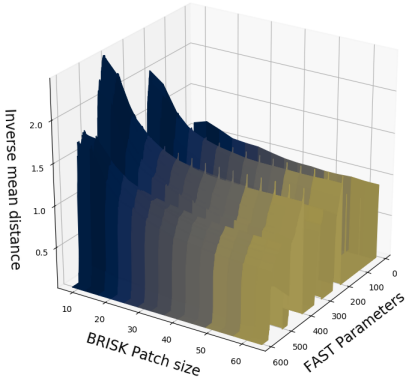
1. **Inverse mean distance of matched descriptors**
   Its main strength is that it can be used entirely without the model estimation stage. It is a fairly **good predictor of keypoint quality** – the maxima are at parameter combinations where both the number of keypoints and projection error are fairly small. However, it **heavily depends on the size of the image patch from which the descriptor is calculated**. The smaller the patch, the more similar descriptors are – this makes mean distance unusable as a metric for optimizing descriptor parameters.
2. **Capped negative mean projection error**

$$e = -\max\left(\frac{1}{|\mathrm{P}|}\sum_{\mathrm{p}\in\mathrm{P}}\left\|\mathrm{H}_{ref}\mathrm{p}-\mathrm{H}_{new}\mathrm{p}\right\|, 500\right) \tag{5.2}$$

   Here P is a grid of points over the image, H_{ref} is the reference homography and H_{new} is found with LMEDS. LMEDS is a nonparametric method and

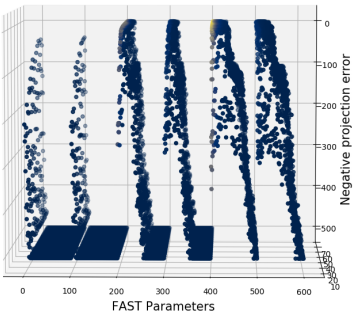


(a) : Keypoint view – maximum at the minimum number of keypoints where precision is still high

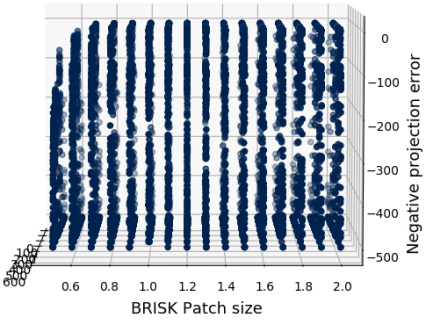


(b) : Descriptor view – distance decreases with patch size

Figure 5.7: Inverse mean distance of matched descriptors



(a) : Keypoint view – precision high when the number of keypoints is large



(b) : Descriptor view – precision falls off for patch sizes smaller than 0.8

Figure 5.8: Capped negative projection error

can thus be used independently of the pipeline preceding it. It requires more than 50% of inliers to be effective, which is not ideal, as there may be better method variants with smaller inlier ratios. Nevertheless, I found it a reasonable compromise, as it allows me to minimize the actual error of point transformations, which is what we want to minimize in tracking. Having the projection error in pixels also makes results easily comprehensible.

The maximum error is capped at 500, which I chose arbitrarily to keep the error for faulty homographies large, but within some bounds – any method with a mean projection error of 500 pixels is entirely unusable anyway.

In all graphs in this section, color signifies computation time: blue (low) to yellow (high).

Comparing pipeline variants

The way I chose to compare the pipeline variants when looking at a range of parameters is to simply loop through the range with some step (e.g. range of (1,100) with a step of 1) and plot the scores for each combination. If I wanted to look at how parameters interact, I tried all combinations with the ranges of all parameters.

Figure 5.8 shows the results when varying parameters for the FAST and BRISK methods. Table 5.3 shows the ranges of parameters. On the plot, keypoint parameters 1-100 correspond to threshold (1-100), nonmaxsuppression off, detector size 5/7; parameters 101-200 correspond to threshold (1-100), nonmaxsuppression on, detector size 5/7, etc.

Method	Parameter	Begin	End	Step
FAST	Threshold	1	100	1
	Nonmaxsuppression	Off	On	1
	Detector size	5/7	9/16	2/4
BRISK	Pattern scale	0.5	2	0.1

Table 5.3: Initial parameter ranges for the FAST keypoint detector and BRISK descriptor extractor methods

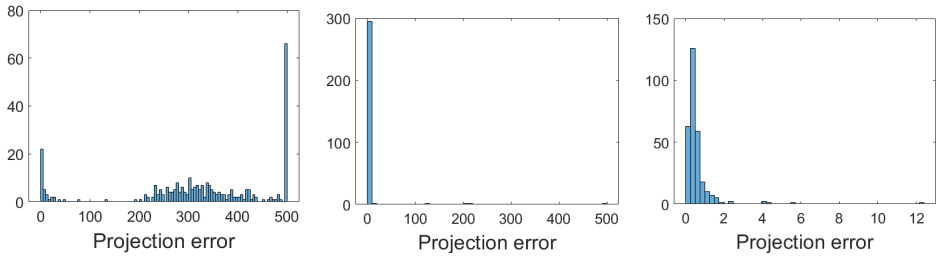
Testing through parameter combinations was much more expensive than to optimize each parameter separately while freezing others at the default value, but provided much more information on how the parameters affect each other.

I considered using something like gradient ascent to get to the local maxima, as it would have saved a lot of resources. However, I didn't use it in the end, for the following reasons:

- The score function is not guaranteed to be smooth with respect to the parameters
- The global maximum may be noisy and not unique – there may be lots of equivalently good parameter combinations that we discard, but may have other advantages (e.g. faster runtime)
- I gain no general insight into the effects of parameter combinations, which can be useful for better understanding what the method parameters do, checking for mistakes, and for later ideas for improvement.

Pipeline stage	Method	Parameter	Initial value	Final value
Preprocess	Median filter	Kernel size	3	3
	CLAHE	Threshold	12	18
		Kernel size	32	8
	Unsharp masking	Kernel size	3	7
Keypoint detector	FAST	Threshold	40	18
		Nonmaxsuppression	On	Off
		Detector size	9/16	9/16
Descriptor extractor	ORB	Patch size	31	N/A
	BRISK	Pattern scale	N/A	1.8
Matching		First/Second ratio	0.9	0.88
Model estimation	RANSAC	Threshold	1	6
		Confidence interval	0.995	0.99
		Max iterations	2000	6000

Table 5.4: Initial and final pipeline methods and parameters



(a) : Initial pipeline errors

(b) : Final pipeline errors

(c) : Final pipeline errors – zoomed in leftmost column

Figure 5.9: Projection error histograms

Hyperparameter search results

When comparing the projection error histograms (Figure 5.9) of the initial and final pipeline on the large dataset, we can see that searching for the optimal parameters and methods was very well worth it. With the initial pipeline, most homographies found were wrong, while with the final parameters there were only a few (3) wrong homographies, and the mean error for the overwhelming majority of cases was within 2 pixels.

The comparison of the initial and final correspondence matching pipeline can be seen in Table 5.4.

5.4 Implementation

Thermocam-PCB is a command-line tool implemented in C++, using the Meson build system. It uses the WIC SDK for communicating with the Workswell Infrared Camera, the OpenCV library for image processing and visualization, and the Crow open-source library for running a webserver.

The thermal image is read using the WIC SDK as 16-bit unsigned raw data, which is in a linear relationship with Celsius temperature values. To visualize it, the 16-bit raw data temperature is recalculated into 8-bit pixels using a fixed range, such that **0 corresponds 15 °C and 255 corresponds to 120 °C**. Thus, if the temperature of the board is below 20 °C, the image is very dark and the tracking feature is likely to fail.

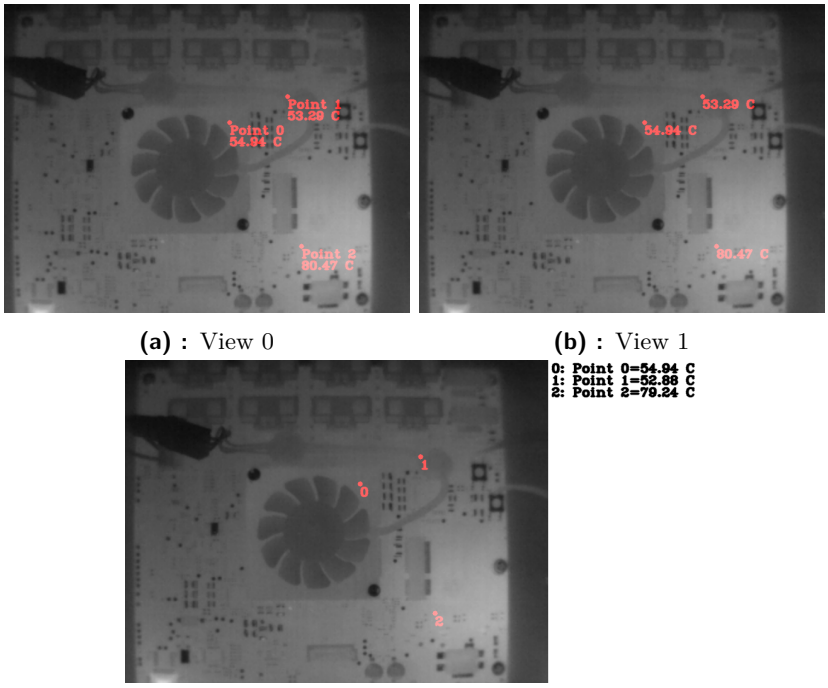
5.4.1 Requirements & Compilation

The requirements for running the tool are:

- License file for the Workswell Infrared Camera
- WIC SDK (installs and uses the eBUS SDK)
- 32/64b x86 Linux, preferably Ubuntu 16.04, for maximum compatibility with the eBUS SDK
- OpenCV 2.4 (default version for Ubuntu 16.04)
- `boost` and `pthread` libraries

To compile the program, run:

```
meson setup build
ninja -C build
```



(a) : View 0 **(b) :** View 1

(c) : View 2

Figure 5.10: The three possible views of Thermocam-PCB

5.4.2 Usage

To simply show the image stream from the WIC, ensure that the camera license file (`.wlic` file) is in the the current directory, and run the application without any arguments:

```
build/thermocam-pcb
```

You can exit the application either by pressing the Esc key, or by pressing Ctrl+C in the terminal.

To enter and save points into a `.json` file, call the application with the `-e` argument (`build/thermocam-pcb -emy-points.json`) and click on the image. After some points have been entered, you can switch between three possible views (shown on Figure 5.10) by pressing Tab.

You can delete entered points with the Backspace button and save your entered points by exiting the program. Saved points can be imported using the `-p` argument, which you can call together with `-e` to edit your saved points.

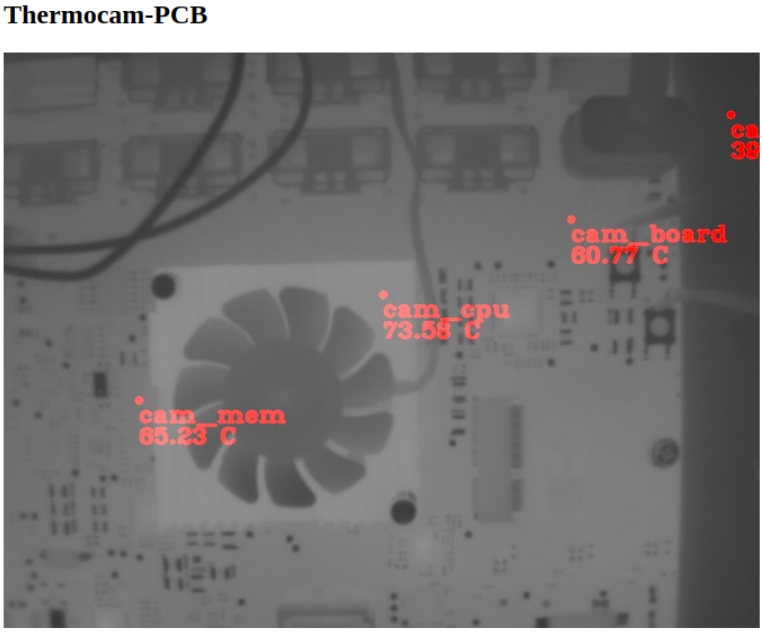


Figure 5.11: Camera image with POI published on the webserver

Saving points also saves the current camera image with them, which the tracker uses as reference. It is recommended that the PCB be turned on when this happens (so the reference image isn't too dark) for better tracking performance.

To turn on the webserver, call the tool with the `-w` argument. The webserver can be accessed on port 8080 – Figure 5.11 shows the web interface of the tool. Querying `ip-address:8080/temperatures.txt` returns the list of POIs and their temperatures in "name=value" format. This is the format that Thermobench understands, thus the two tools can be used together.

It is possible to monitor tracking performance, by querying `ip-address:8080/position-std.txt`, which returns the mean rolling standard deviations of POI positions. During normal operation, these will be below 2-3 pixels, however, if tracking fails, the points will move around randomly on the image and the standard deviations will be really high.

5.4.3 Command line reference

```
Usage: thermocam-pcb [OPTION...] [--] COMMAND...
Displays thermocamera image and entered points of interest and their
temperature. Writes the temperatures of entered POIs to stdout.

  -c, --csv-log=FILE         Log temperature of POIs to a csv file instead of
                             printing them to stdout.
  -d, --delay=NUM            Set delay between each measurement/display in
                             seconds.
  -e, --enter-poi[=FILE]     Enter Points of interest by hand, optionally save
                             them to json file at supplied path.
  -l, --license-dir=FILE     Path to directory containing WIC license file.
                             "." by default.
  -p, --poi-path=FILE        Path to config file containing saved POIs.
  -r, --record-video=FILE    Record video and store it with entered filename
  -s, --show-poi=FILE        Show camera image taken at saving POIs.
      --save-img-dir=FILE    Target directory for saving an image with POIs
                             every "save-img-period" seconds.
                             "." by default.
      --save-img-period=NUM  Period for saving an image with POIs to
                             "save-img-dir".
                             1s by default.
  -t, --track-points         Turn on tracking of points.
  -v, --load-video=FILE      Load and process video instead of camera feed
  -w, --webserver            Start webserver to display image and
                             temperatures.
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Requires path to directory containing WIC license file to run with camera.

Controls:
Tab                - Change view  (Full | Temperature only | Legend)
Mouse click (left) - Enter point  (only with --enter-poi)
Backspace          - Remove point (only with --enter-poi)
Esc                - Exit program

Report bugs to https://github.com/CTU-IIG/thermocam-pcb/issues.
```

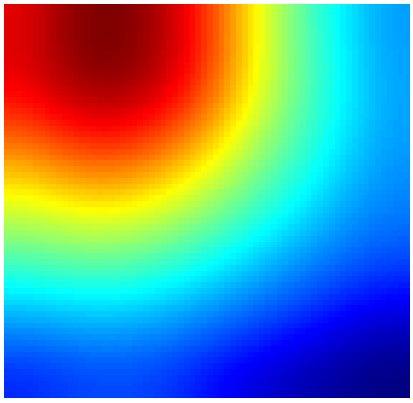


Figure 5.12: Smoothed heatmap of a single frame from GPU 32-bit float addition

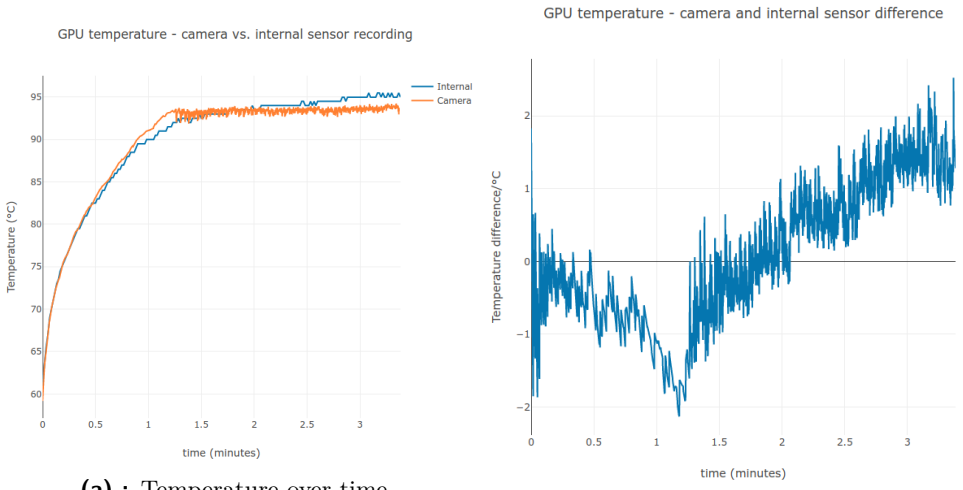
5.5 Results

If we compare the functional requirements of the tool established in section 5.1.1 to the implementation described in the previous section, we can see that all the functional requirements of the tool have been met. Thus, this section only discusses how Thermocam-PCB meets its quality requirements, established in section 5.1.2:

1. **Precise temperature measurement (±0.5°C)**
   The WIC specs state a general accuracy of ±2 °C. However, the WIC SDK documentation [Jer17] states that measurement accuracy can be improved by calibrating the camera using Flat Field Correction(FFC). We set the camera to do FFC periodically every minute to ensure precise measurement – the FFC takes 1-2 seconds, thus it seemed like a reasonably small period to me.

   To test the accuracy of thermocamera measurements, we ran a GPU benchmark (32-bit float addition) while simultaneously recording with Thermobench and Thermocam-PCB.

   For simplicity, we manually selected a point which seemed to lie in the middle of the heat source (GPU) on the thermal image (see Figure 5.12), which was smoothed to reduce noise. Figure 5.13a shows the temperature of that point over time, aligned in time with the GPU temperature from Thermobench. We can see that the plots are pretty close all along, but we can see the magnitude of the error more clearly on Figure 5.13b. The error stays within 2 °C, and is smaller for a significant part of the measurement.



(a) : Temperature over time

(b) : Temperature difference

Figure 5.13: Thermobench & Thermocam-PCB measurement comparison

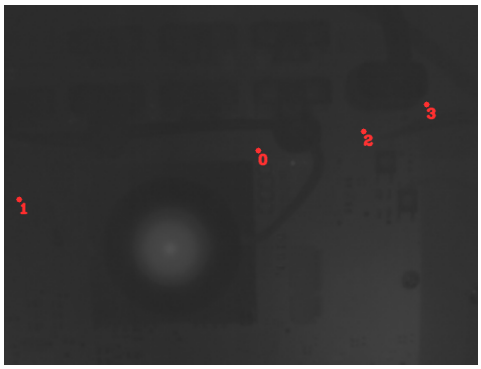
The differences can be explained not only by the inaccuracy of the camera – the GPU temperature sensor may be located somewhere else than the location we chose. However, even with this simple setup, we can see that the difference between the two measurements is small.

We did not manage to reduce measurement error from 2 °C to 0.5 °C, however, from the previous example it is evident that the camera measurements can be used in conjunction with and to complement internal sensor measurements.

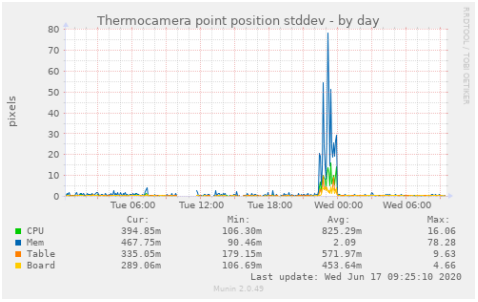
2. **Precise point tracking (max 2-3px error)**
As discussed in section 5.3.3, when tested on a large set of transformed images with various local temperature changes, the overwhelming majority of homographies found had mean projection errors below 2 pixels. Thus, this constraint is satisfied for most used temperatures, and only fails on temperatures of around 15-20 °C where the image quality suffers considerably.

3. **Real-time computation on an embedded platform (<400 ms)**
Although Thermocam-PCB works with a computational time of around 350 ms, which allows it to service around every 3rd frame. This makes it real-time by our definition, so this requirement is satisfied.

4. **Detectable & recoverable failure modes**
The main failure mode of the algorithm is when the input images get so dark that there is very little image depth to select good features (see Figure 5.14a). As input images are treated as independent, recovery from this state is implicitly in the algorithm, when the PCB heats up a bit. In this state, the homographies found exhibit high error. As most of the time the PCB will be static, we can monitor this through plotting



(a) : Image too dark for proper function



(b) : Standard deviation of point positions over a one day period.
The PCB got too cold and thus the image too dark Tuesday 23:00-00:00.

Figure 5.14: Failure mode for tracking

the rolling standard deviations of point positions, as seen in Figure 5.14b. With these in place, failure is both detectable and recoverable.

5. **Robustness to local & global change of temperature over surfaces with variable emissivity**
The performance of the solution on the larger dataset (see section 5.3.3), which has transformed images of various global and local temperatures is satisfactory, discussed in point 2. The solution fails for very low temperatures, where image quality degrades, discussed in point 4. However, those temperatures only occur when the board is turned off, which is not a state in which we are interested in measuring.

Chapter 6

Determining heat sources on chip for different workloads

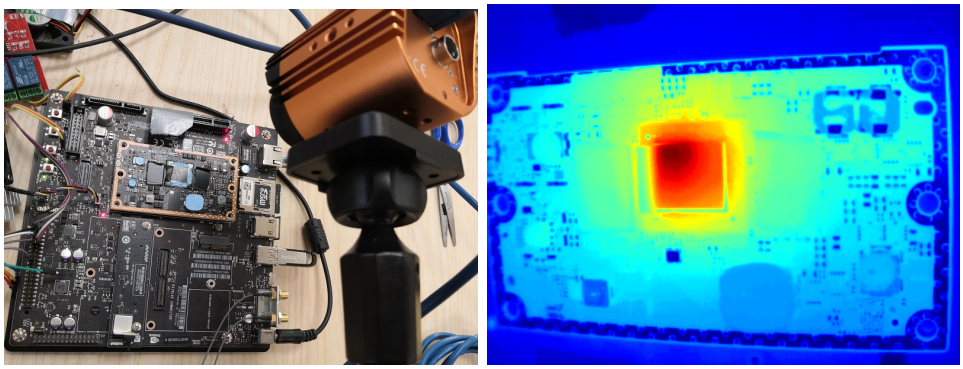
When we design algorithms to lower chip temperature, it is useful to know which parts of the chip heat up more for which workloads. Knowing the locations of these heat sources allows us to heat up the chip more uniformly (e.g. by combining workloads in the right way), ensuring better heat spread and lower overall temperature.

This chapter describes a method for determining on-chip heat sources from videos of a chip being under a single type of workload – CPU arithmetic, GPU arithmetic, CPU-based memory-bound. After determining heat sources for each workload separately, and we can compare them and classify them as workload-specific, results of noise, or from an area generally used by all workloads.

The method used to locate heat sources in this chapter is inspired by the one described in [SZA+19]. However, we use Gaussian smoothing instead of discrete cosine transform for noise filtering and histogram maxima instead of k-means for determining heat source locations (detailed in section 6.4).

6.1 Hardware setup

In this chapter we use an NVIDIA Jetson TX2 (see section 2.1) with its heat sink removed to run our workloads during the experiments. We positioned



(a) : WIC looking at the Jetson board, chips covered with black insulating tape

(b) : Image of the Jetson board captured by the WIC, visualized with the JET colormap

Figure 6.1: Hardware setup – external view and view from WIC

the WIC thermal camera (see section 2.2) to look at the Jetson board with the TX2 System-on-Chip (SoC) and two RAM chips centered in the picture. The camera was looking at the board from a slightly tilted angle to reduce thermal reflections and from as close a range as the it could keep focus.

The SoC and the two RAM chips were covered with black insulating tape to increase their thermal emissivity to approximately 0.95. High emissivity is essential for both recording precise temperatures and detecting small differences in on-chip temperature. The setup and an example image captured with the setup are shown in Figure 6.1.

6.2 Experiment setup

The experiments were set up to answer the following questions:

1. Is it possible to locate individual CPU cores using the thermal recordings?
2. Is it possible to locate GPU streaming multiprocessors using the thermal recordings?
3. Is is possible to tell the difference between L1 cache/L2 cache/main memory accesses when looking at the SoC?
4. How does memory traffic influence DRAM chip temperature?

Arithmetic benchmark	Runs on	
32bit floating point addition	ARM Cortex-A57 CPU	core 0
		core 1
		core 2
		core 3
	NVIDIA Denver 2	core 0
		core 1
	All CPU cores	
32bit integer addition	ARM Cortex-A57 CPU	core 0
		core 1
		core 2
		core 3
	NVIDIA Denver 2	core 0
		core 1
	All CPU cores	
32bit floating point addition	NVIDIA Pascal GPU	

Table 6.1: Arithmetic benchmarks run for determining heat sources

Memory benchmark		Runs on
Memory sequential access	16KB (L1 cache)	ARM Cortex-A57 CPU core 0
	512KB (L2 cache)	
	12M (Main memory)	
Memory random access	16KB (L1 cache)	
	512KB (L2 cache)	
	12M (Main memory)	

Table 6.2: Memory benchmarks run for determining heat sources

To answer these questions, we selected a small set of arithmetic and memory benchmarks, shown in Tables 6.1 and 6.2. We used both Thermobench and Thermocam-PCB for measurement – Thermobench ran the benchmarks and recorded on-chip sensor readings, and Thermocam-PCB recorded a thermal video of the benchmark run. Only the thermal videos were used to determine heat source locations.

The TX2 was left to cool down between experiments until the Cortex-A57 on-chip sensor showed 55 °C, which was the idle CPU temperature. This ensured the independence of benchmark runs.

6.3 Theoretical basis

6.3.1 The heat diffusion equation

We use the equation for heat diffusion in solids [BLID17] as the basis for determining heat source locations:

$$\nabla^2 T + \frac{\dot{q}}{k} = \frac{1}{\alpha}\frac{\partial T}{\partial t} \tag{6.1}$$

Here, T is thermodynamic temperature, $\dot{q}$ is the rate of thermal energy generation in the medium, k is thermal conductivity, α is thermal diffusivity, and t is time.

Heat sources are locations of local heat production maxima on chip – thus we are looking for local maxima of $\dot{q}$:

$$\dot{q} = k\left(-\nabla^2 T + \frac{1}{\alpha}\frac{\partial T}{\partial t}\right) \tag{6.2}$$

The locations of heat sources (x_h, y_h) are thus:

$$(x_h, y_h) = \arg\max \dot{q}(x, y) = \arg\max k\left(-\nabla^2 T(x, y) + \frac{1}{\alpha}\frac{\partial T(x, y)}{\partial t}\right) \tag{6.3}$$

We assume thermal conductivity, k, to be constant across the chip and in time, thus multiplying with it does not affect the location of maxima:

$$(x_h, y_h) = \arg\max \dot{q}(x, y) = \arg\max \left(-\nabla^2 T(x, y) + \frac{1}{\alpha}\frac{\partial T(x, y)}{\partial t}\right) \tag{6.4}$$

When looking at a chip with a thermal camera, we know the thermodynamic temperature at each pixel position for each frame, thus we can calculate both the Laplacian and the time derivative. Thus the only unknown parameter is α – thermal diffusivity.

6.3.2 Determining thermal diffusivity

The thermal diffusivity of CMOS chips depends on the thickness of the various layers of a chip, and their thermal properties [KM09]. These parameters are usually only known to the manufacturer, and vary between chips. Thus, it is hard to get a precise estimate of diffusivity from tables of material constants.

Studies aimed at measuring thermal diffusivity on CMOS chips ([KM09] [Ebr70]) use specialized on-chip hardware (an electrothermal filter), which is not available to us.

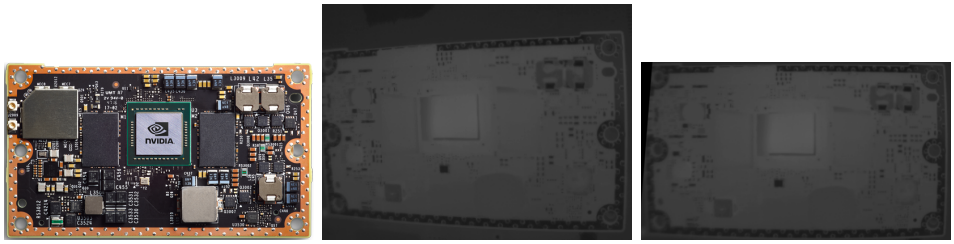
Our idea for calculating thermal diffusivity from a series of thermal images is heating up the chip with some workload and recording how the chip cools down after the workload has ended. With processor workloads the switch off is basically instant compared to the temperature change, and with no heat source, the heat diffusion equation changes to:

$$\nabla^2 T(x,y) = \frac{1}{\alpha}\frac{\partial T(x,y)}{\partial t} \tag{6.5}$$

$$\alpha = \frac{\frac{\partial T}{\partial t}}{\nabla^2 T} \tag{6.6}$$

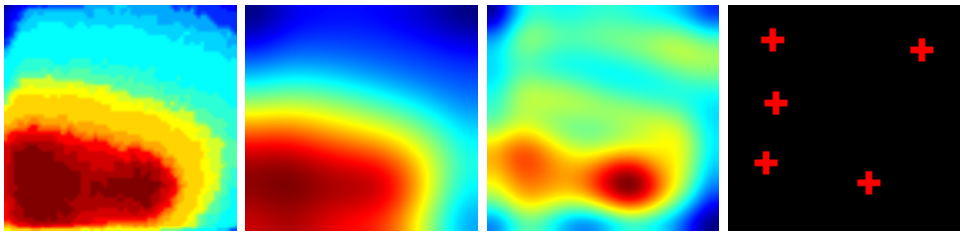
However, when calculated for all pixels in all frames of the chip cooldown video with this method, we get wildly different values for α varying in both time and space. Thus we ignore the temporal parameter altogether (just as described in [SZA+19]), as adding it would probably do more harm than good. We thus use the local maxima of only the negative Laplacian to find the heat source locations, and videos where the chip started from cold and the workload was applied for the entire duration.

$$(x_h, y_h) = \arg\max -\nabla^2 T(x,y) \tag{6.7}$$



(a) : Reference **(b) :** Recorded **(c) :** Normalized

Figure 6.2: Normalization of the tilted image to top-down view



(a) : Original thermal image **(b) :** Smoothed thermal image **(c) :** Negative Laplacian **(d) :** Negative Laplacian local maxima

Figure 6.3: Heat source detection on the TX2 SoC during the CPU 32bit integer addition benchmark with all cores active

6.4 Implementation

As the videos were recorded with the camera tilted at a slight angle, we normalize them with a homography to a top-down view using a reference image (see Figure 6.2). The thermal and reference grayscale images are too different to find the homography automatically, so we calculate the homography using manually selected corresponding points.

After normalizing for tilt we create 3 types of videos: with the SoC, left RAM chip and right RAM chip cropped out, for each benchmark measured.

Before applying the Laplacian on the cropped video images, they need to be smoothed – the Laplacian is the second spatial derivative of the image, and numerical derivatives are highly sensitive to noise. Thermal video recordings suffer from both temporal and spacial noise [Ken93], so we apply a Gaussian kernel of size 7 to the video in both the two spacial directions, and time – see Figure 6.3b.

For each frame we detect the local maxima of the negative Laplacian over

the image (Figures 6.3c, 6.3d), sum the locations of of these maxima over the video frames into a histogram, and divide with the number of video frames. This results in a 2D histogram of heat sources with each frame essentially "voting" for heat source locations. The histogram bin heights thus only show how consistent the maxima locations are through frames, it does not show how large the maxima are.

6.5 Experiment results

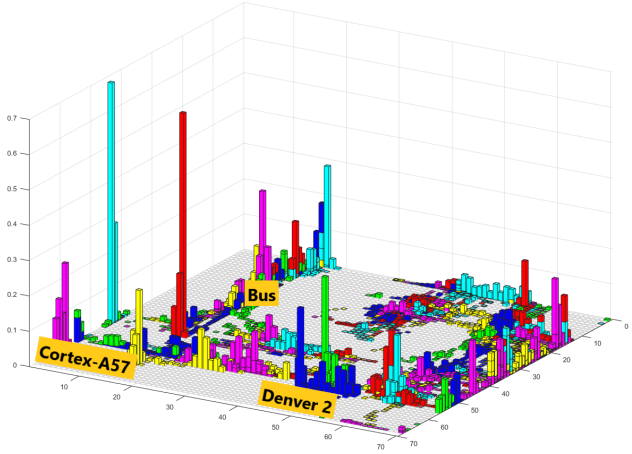
Let's use the results of benchmark runs to answer the questions outlined in section 6.2:

1. **Is it possible to locate CPU/GPU cores using the thermal recordings?**
   Figure 6.4 shows the combined histogram plots of the CPU arithmetic benchmarks run on the 6 CPU cores of the SoC. We can see on the left side of the plot that the histogram maxima for the Cortex-A57 benchmarks form a precise rectangle, clearly showing the positions of the processor cores. The histogram maxima for the Denver benchmarks are separated from the Cortex-A57 ones, and more to the right, but are close to each other, so it is highly likely that they show the positions of the Denver cores.

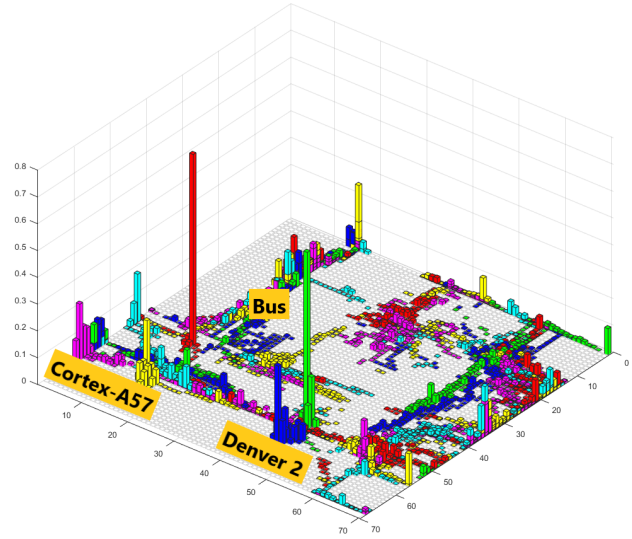
   All 6 maxima are stable across the two types of instruction runs, so we can assume that they show real heat sources on the chip. We can assume that some lines with smaller amplitudes that are stable through the two types of instruction benchmarks show the locations of buses for the relevant processors.

   In conclusion, it is definitely possible to locate CPU cores on the SoC, even each one separately.

2. **Is it possible to locate GPU streaming multiprocessors using the thermal recordings?**
   Figure 6.5 shows the histogram plot for the GPU benchmark run. We can clearly see two maxima, which is the number of streaming multiprocessors (each with 128 cores) in the TX2 Pascal GPU, so it is highly likely that the maxima show their locations. Thus, we can easily locate the GPU multiprocessors.



(a) : 32bit floating point addition



(b) : 32bit integer addition

Figure 6.4: Combined heat source histogram plots of CPU float/int benchmarks
Red, Cyan, Yellow, Magenta = Cortex-A57 core 0,1,2,3
Green, Blue = Nvidia Denver 2 core 0,1

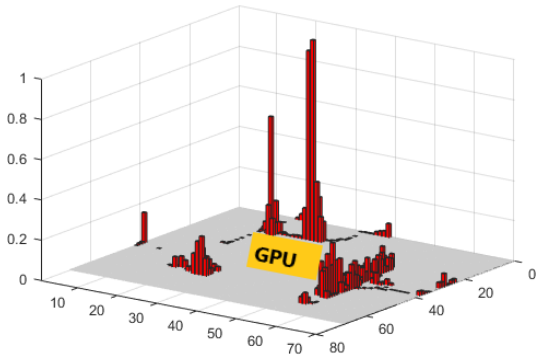


Figure 6.5: Heat source histogram plot of the GPU arithmetic benchmark run

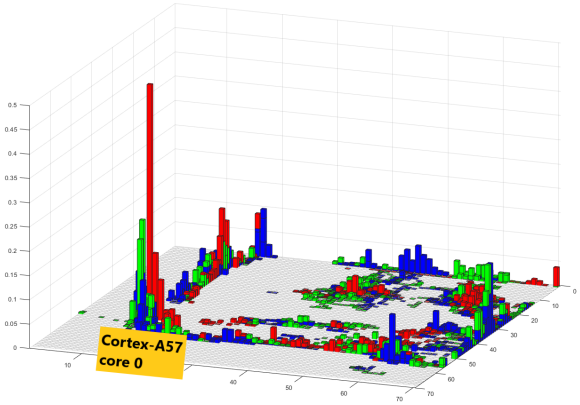


Figure 6.6: Heat source histogram plot of random memory access benchmarks on the SoC
Red = L1 cache, Green = L2 cache, Blue = Main memory

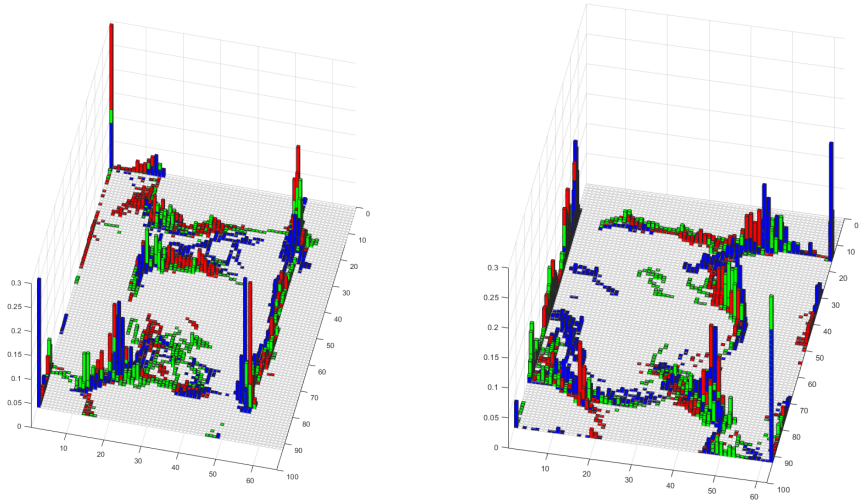
Benchmark	SoC	RAM 0	RAM 1
L1 cache	57.32	49.37	49.21
L2 cache	57.34	49.88	50.30
Main memory	58.26	51.15	51.16

Table 6.3: Mean temperatures of chips at the end of random access benchmark runs in °C

3. **Is is possible to tell the difference between L1 cache/L2 cache/main memory accesses when looking at the SoC?**
   Figure 6.6 shows the shows the combined histogram plots for the 3 random array access benchmark runs. There is no significant difference between the three plots, and the main heat source visible on the plot is the Cortex-A57 core 0, on which the benchmark ran. Thus, at least using our method in its current form, it is not possible to tell the difference between these benchmark runs just by looking at the SoC.

4. **How does memory traffic influence DRAM chip temperature?**
   Figure 6.6 shows the histogram plots for the two RAM chips. We can again see that there is no significant difference between the 3 benchmarks. As the benchmarks using only L1 and L2 caches write to main memory very rarely, if the RAM chips were significant hotspots, we should see some difference.

   Table 6.3 shows the mean temperatures of chips at the end of random access memory benchmark runs. We can see that the RAM chips are indeed warmer at the end of the benchmark run that accesses main memory, and L1 and L2 final temperatures are very similar. However, the SoC also gets warmer during the main memory benchmark, and it is thus unclear if the RAM temperature difference is caused by the RAM chip itself, or by the SoC heating up its surroundings.



(a) : Left RAM chip

(b) : Right RAM chip

Figure 6.7: Heat source histogram plots of random memory access benchmarks on RAM chips
Red = L1 cache, Green = L2 cache, Blue = Main memory

6.6 Conclusion

In this chapter, we outlined a basic method of detecting on-chip heat sources. Our solution used the heat diffusion equations and ignored its transient component, which probably made it more inaccurate. During the run of benchmarks there was no cooling applied to the chips, except for maybe some passive room temperature air flow, making localization harder as the heat spread further and faster on the chip. Despite these factors, we managed to clearly localize CPU processor cores and GPU streaming multiprocessors.

These results imply that if we manage to improve our model by correctly calculating heat diffusivity and apply cooling to the chips (for example backside cooling with a Peltier device, similarly as described in [SZA+19]), we can likely accurately detect other, smaller heat sources too and can thus develop more sophisticated algorithms for managing chip temperature.

It would also be worthwhile to run the main memory random access experiment again with a larger memory array, to see if that changes the histogram plots of the RAM chips. It would also be worth redoing the memory experiments with cooling applied to the SoC, to eliminate the effect of heat spread from the SoC to the RAM chips and thus check if the RAM chips heat up more by themselves.

Chapter 7

Temperature reduction methods

Temperature reduction without compromising workload performance is one of the major goals of the THERMAC project, described in section 1. In this chapter we propose and evaluate two methods for reducing temperature at equal performance.

We measure and evaluate the effects of **Compiler optimization levels** and **Frequency scaling** on temperature and performance.

We test these methods on the following workloads:

1. Software 3D renderer
2. KCF object tracking algorithm (section 4.3.1)
3. KLT object tracking algorithm (section 4.3.2)

7.1 Experiment setup

We used the NVIDIA TX2 platform (see section 2.1) for our measurements. The cooler on the System on Chip (SoC) is removed to amplify the temperature differences between the various compiler/frequency settings.

We use standalone open-source implementations of the workloads – [Sok20] for the 3D software renderer, [pby16] for the KCF tracker and [Bir07] for the KLT tracker. These are modified to only read the input data once from disk, and then run their calculations on the same data in an infinite loop. This way their runtime can easily be controlled by Thermobench. The benchmarks track how many times they have done their work and periodically print the current aggregate value, which is recorded by Thermobench for later analysis.

We run our workloads on four optimization levels: O0-O3. For frequency scaling, we run our tests on the Cortex-A57 core 0, setting its frequency from 652-1421 MHz, with a step of roughly 150 MHz. To preserve the independence of measurements, all measurements start out at the temperature of 58 °C and the chip is left to cool down to that temperature between tests.

7.2 Results

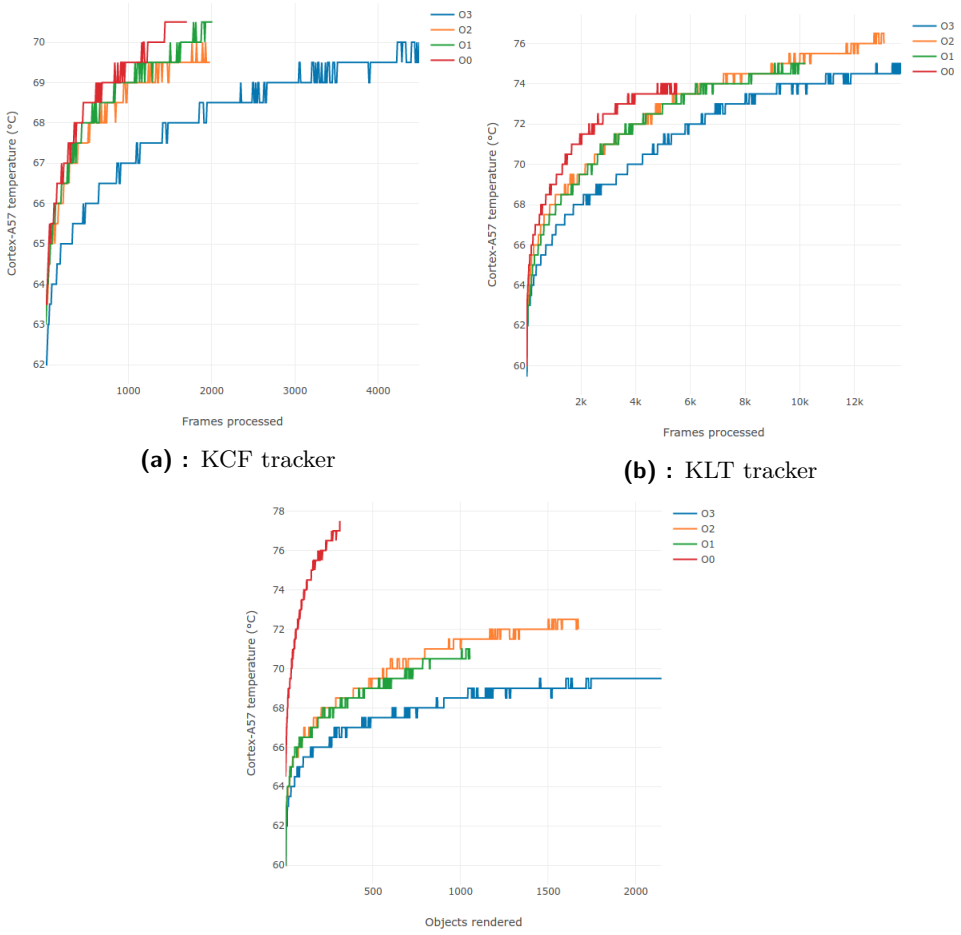
7.2.1 Compiler optimization level

Figure 7.1 shows the change in temperature vs. the work done (objects rendered for the renderer, frames processed for the trackers) for the three workloads. We can see on all 3 graphs, that in terms of work done per temperature increase:

1. The O0 optimization level is worse than any other
2. O1, O2 are roughly equivalent
3. O3 is much better than any other optimization level

This is an important finding, as O2 is still commonly used, and often prioritized above O3 – this set of tests makes it very clear that in terms of temperature increase per unit of performance for these workloads, O3 is very highly preferable.

It would be worth testing if the better performance of O3 is caused mainly by a specific optimization (e.g. loop vectorization), or by their combination.



(a) : KCF tracker

(b) : KLT tracker

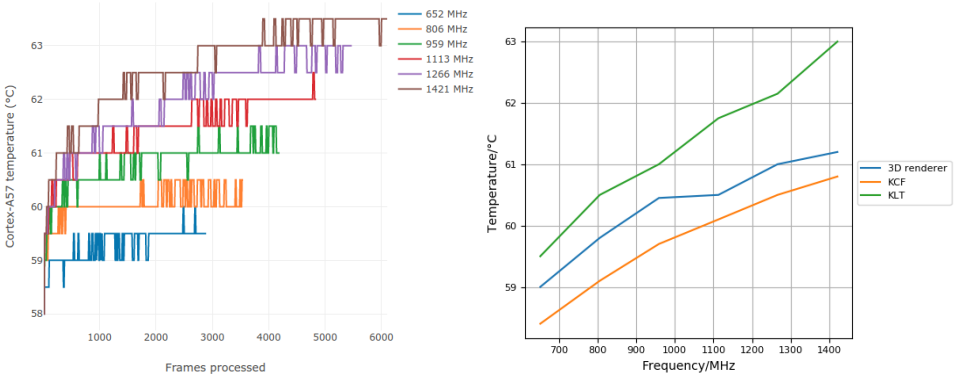
(c) : 3D software renderer

Figure 7.1: Compiler optimization level comparison –Temperature vs. work done

7.2.2 Frequency scaling

For all workloads, the curves for temperature increase versus work done follow a simple rule: the larger the frequency, the higher the temperature per unit work done (see Figure 7.2a). Because of this, we can better visualize the relationship between frequency and temperature per unit work done by plotting the temperature for each frequency at equal work done – see Figure 7.2b.

We compare the benchmark runs at the maximum amount of work done on the smallest frequency, as that is the point closest to steady-state temperature where the comparison is still fair.



(a) : KLT tracker
Cortex-A57 core 0 frequency

(b) : Temperature at equal work done

Figure 7.2: Frequency scaling comparison

We can see from Figure 7.2b, that for the KLT tracker, the temperature with equal work done has a roughly linear relationship. However, for the KCF tracker and renderer, the temperature increases slower and slower as the frequency rises – thus, it is more worth running these workloads on higher frequencies. It is not clear however, that this effect is not due to the maximum temperature for the KLT tracker being higher than the rest – it would be worth to test that in another set of experiments.

7.3 Temperature reduction methods

Given the results of section 7.2, we propose the following methods to reduce the SoC temperature:

1. **Use the optimization level O3**
   In section 7.2.1 it was clearly shown that in terms of temperature change per unit performance, the optimization level O3 significantly outperforms all other optimization levels.

2. **Use higher frequencies for the same workload**
   In section 7.2.2 we have seen that the temperature for equal work done increases more and more slowly for increasing frequency (KCF tracker and 3D renderer) or proportionally with increasing frequency (KLT tracker). Thus, it is a good idea to use higher frequencies to calculate the same workload as it results in a smaller relative temperature increase.

Chapter 8

Conclusion

In summary, the following was achieved in this thesis:

1. **Thermobench tool & benchmarks**
   I have developed the basis for the Thermobench tool described in chapter 3, which has been later expanded by other THERMAC team members – Michal Sojka, Ondřej Benedikt, Alexander Barinov. It has low impact on temperature and performance, and among other features is capable of reading internal sensors and measuring CPU usage.

   Thermobench can also communicate with Thermocam-PCB by querying point temperatures from the webserver built into Thermocam-PCB.

   In addition to the tool I developed several benchmarks, and executed them on the TX2 platform – the relevant experiments are discussed in section 3.4. The same benchmarks and experiments can easily be run on other ARM platforms using the Thermobench tool. They provide useful information for instruction usage in arithmetic-bound applications.

2. **Thermocam-PCB tool**
   I have developed the Thermocam-PCB tool – see chapter 5. It achieves most of its requirements – see section 5.5. It is a useful tool for external measurement of point temperatures on the PCB, as well as providing useful information through its thermal image.

3. **Locating heat sources**
   I have proposed a method for locating heat sources on-chip, and used this method and the Thermocam-PCB tool to identify the locations of major heat sources (CPU cores and GPU multiprocessors) on the NVIDIA TX2 SoC.

4. **Temperature reduction methods**
   I have proposed two software-based temperature reduction methods (compiler optimization level choice and frequency scaling) and used the Thermobench tool to measure and evaluate their effectiveness on object tracking and software 3D rendering workloads.

The tools constructed and results achieved will be further used and expanded in the THERMAC project.

Appendix A

Bibliography

[AN17] C.S. Asha and A.V. Narasimhadhan, *Robust infrared target tracking using discriminative and generative approaches*, Infrared Physics & Technology **85** (2017), 114–127.

[and17] and, *Infrared thermography sensor for temperature and speed measurement of moving material*, Sensors **17** (2017), no. 5, 1157.

[BB17] Ertugrul Bayraktar and Pinar Boyraz, *Analysis of feature detector and descriptor combinations with a localization experiment for various performance metrics*, CoRR **abs/1710.06232** (2017).

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, *Julia: A fresh approach to numerical computing*, SIAM review **59** (2017), no. 1, 65–98.

[Bir07] Stan Birchfield, *KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker*, August 2007.

[BLID17] T.L. Bergman, A.S. Lavine, F.P. Incropera, and D.P. DeWitt, *Fundamentals of heat and mass transfer*, Wiley, 2017.

[CH14] Tze-Yuan Cheng and Cila Herman, *Motion tracking in infrared imaging for quantitative medical diagnostic applications*, Infrared Physics & Technology **62** (2014), 70–80.

[CLmS16] Lu Chaoliang, Ma Lihua, Yu min, and Cui Shumin, *Regional information entropy demons for infrared image nonrigid registration*, Optik **127** (2016), no. 1, 227–231.

[DC19] Z. Dong and L. Chen, *Image registration in pcb fault detection based on infrared thermal imaging*, 2019 Chinese Control Conference (CCC), 2019, pp. 4819–4823.

[DT05] N. Dalal and B. Triggs, *Histograms of oriented gradients for human detection*, 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 1, 2005, pp. 886–893 vol. 1.

[Ebr70] J Ebrahimi, *Thermal diffusivity measurement of small silicon chips*, Journal of Physics D: Applied Physics **3** (1970), no. 2, 236–239.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener, *TACLeBench: A benchmark collection to support worst-case execution time research*, 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (Dagstuhl, Germany) (Martin Schoeberl, ed.), OpenAccess Series in Informatics (OASIcs), vol. 55, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[HCMB14] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista, *High-speed tracking with kernelized correlation filters*, CoRR **abs/1404.7584** (2014).

[HGMLHM12] Rafael Hidalgo-Gato, Patricia Mingo, José Miguel López-Higuera, and Francisco J. Madruga, *Pre-processing techniques of thermal sequences applied to online welding monitoring*, Quantitative InfraRed Thermography Journal **9** (2012), no. 1, 69–78.

[HZ04] R. I. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, second ed., Cambridge University Press, ISBN: 0521540518, 2004.

[JC16] C. Y. Jeong and S. Choi, *A comparison of keypoint detectors in the context of pedestrian counting*, 2016 International Conference on Information and Communication Technology Convergence (ICTC), 2016, pp. 1179–1181.

[Jer17] Jan Jerabek, *Wic sdk documentation - version for linux*, Jan 2017.

[JM16] Ondra Chum Jiří Matas, *Tracking with correlation filters*, 2016, Slide 58.

[JM20a] _____, *Local feature extraction and description for wide-baseline matching, object recognition and image retrieval methods, stitching and more*, 2020, Slide 45.

[JM20b] _____, *Local feature extraction and description for wide-baseline matching, object recognition and image retrieval methods, stitching and more*, 2020, Slide 69.

[JM20c] _____, *Local feature extraction and description for wide-baseline matching, object recognition and image retrieval methods, stitching and more*, 2020, Slide 72.

[JM20d] _____, *Local feature extraction and description for wide-baseline matching, object recognition and image retrieval methods, stitching and more*, 2020, Slide 58.

[JM20e] _____, *Robust model estimation from data contaminated by outliers*, 2020, Slide 13.

[JPR+07] Andreja Jarc, Janez Pers, Peter Rogelj, Matej Perše, and Stanislav Kovačič, *Texture features for affine registration of thermal (flir) and visible images.*

[Ken93] Howard V. Kennedy, *Modeling noise in thermal imaging systems*, Infrared Imaging Systems: Design, Analysis, Modeling, and Testing IV (Gerald C. Holst, ed.), vol. 1969, International Society for Optics and Photonics, SPIE, 1993, pp. 66 – 77.

[KLB+17] Andriy Guilherme Krefer, Maiko Min Ian Lie, Gustavo Benvenutti Borba, Humberto Remigio Gamba, Marcos Dinís Lavarda, and Mauren Abreu de Souza, *A method for generating 3d thermal models with decoupled acquisition*, Computer Methods and Programs in Biomedicine **151** (2017), 79–90.

[KM09] S. M. Kashmiri and K. A. A. Makinwa, *Measuring the thermal diffusivity of cmos chips*, SENSORS, 2009 IEEE, 2009, pp. 45–48.

[KWW+14] Lai Kang, Lingda Wu, Yingmei Wei, Bing Yang, and Hanchen Song, *A highly accurate dense approach for homography estimation using modified differential evolution*, Engineering Applications of Artificial Intelligence **31** (2014), 68–77.

[LCS11] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart, *Brisk: Binary robust invariant scalable keypoints*, Proceedings of the 2011 International Conference on Computer Vision (USA), ICCV '11, IEEE Computer Society, 2011, p. 2548–2555.

[Low99] D. G. Lowe, *Object recognition from local scale-invariant features*, Proceedings of the Seventh IEEE International Conference on Computer Vision, vol. 2, 1999, pp. 1150–1157 vol.2.

[Low04] David G. Lowe, *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision **60** (2004), no. 2, 91–110.

[LYC+19] Jinyu Li, Bangbang Yang, Danpeng Cheng, Wang Nan, Guofeng Zhang, and Hujun Bao, *Survey and evaluation of monocular visual-inertial slam algorithms for augmented reality*, Virtual Real. Intell. Hardw. **1** (2019), 386–410.

[LYZ+14] Xiangyun Liao, Zhiyong Yuan, Qi Zheng, Qian Yin, Dong Zhang, and Jianhui Zhao, *Multi-scale and shape constrained localized region-based active contour segmentation of uterine fibroid ultrasound images in hifu therapy*, PloS one **9** (2014), e103334.

[Mod11] M. Moderhak, *FFT spectra based matching algorithm for active dynamic thermography*, Quantitative InfraRed Thermography Journal **8** (2011), no. 2, 239–242.

[NVI20] NVIDIA Corporation, *Jetson tx2 developer kit*, 2020, [Online; accessed July 27, 2020].

[NXP20] NXP Semiconductors, *Mcimx8qm-cpu: i.mx 8quadmax multisensory enablement kit (mek)*, 2020, [Online; accessed July 27, 2020].

[Ort12] Raphael Ortiz, *Freak: Fast retina keypoint*, Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (USA), CVPR '12, IEEE Computer Society, 2012, p. 510–517.

[pby16] pby5, *kcf tracker*, December 2016.

[RLSB19] Abolfazl Irani Rahaghi, Ulrich Lemmin, Daniel Sage, and David Andrew Barry, *Achieving high-resolution thermal imagery in low-contrast lake surface waters by aerial remote sensing and image registration*, Remote Sensing of Environment **221** (2019), 773–783.

[Rub20] Rubicon Communications, LLC, *Minnowboard turbot dual core board*, 2020, [Online; accessed July 27, 2020].

[SC19] A. A. Sarawade and N. N. Charniya, *Detection of faulty integrated circuits in pcb with thermal image processing*, 2019 International Conference on Nascent Technologies in Engineering (ICNTE), 2019, pp. 1–6.

[SES10] Tobias Senst, Volker Eiselein, and Thomas Sikora, *Ii-lk – a real-time implementation for sparse optical flow*, Image Analysis and Recognition (Berlin, Heidelberg) (Aurélio Campilho

and Mohamed Kamel, eds.), Springer Berlin Heidelberg, 2010, pp. 240–249.

[Sok20] Dmitry V. Sokolov, *Tiny Renderer or how OpenGL works: software rendering in 500 lines of code*, January 2020.

[SZA+19] S. Sadiqbatcha, H. Zhao, H. Amrouch, J. Henkel, and S. X. . Tan, *Hot spot identification and system parameterized thermal modeling for multi-core processors through infrared thermal imaging*, 2019 Design, Automation Test in Europe Conference Exhibition (DATE), 2019, pp. 48–53.

[Thi17] Nicolas Huynh Thien, *Dataset augmentation with random homographies*, Sep 2017.

[TK91] Carlo Tomasi and Takeo Kanade, *Detection and tracking of point features*, Tech. report, International Journal of Computer Vision, 1991.

[TLF10] E. Tola, V. Lepetit, and P. Fua, *Daisy: An efficient dense descriptor applied to wide-baseline stereo*, IEEE Transactions on Pattern Analysis and Machine Intelligence **32** (2010), no. 5, 815–830.

[Wik20] Wikipedia, the free encyclopedia, *Epipolar geometry*, 2020, [Online; accessed June 13, 2020].

[Wor20] Workswell s.r.o., *Workswell infrared camera (wic)*, 2020, [Online; accessed July 27, 2020].