Bachelor Thesis

Power and Energy Efficiency Analysis of HPC Workloads on Modern CPU Architectures

Thomas Klotz

supervised by

Philipp Gschwandtner, PhD.

Leopold-Franzens-Universität Innsbruck
Faculty of Mathematics, Computer Science and Physics

14th of March, 2023

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

14.03.2023	[illegible]
Datum	Unterschrift

Abstract

Modern CPUs feature complex mechanisms in order to manage the trade-off between performance and energy, such as DVFS or power capping. Part of the data used to drive these mechanisms is available to the user, enabling detailed analyses of the power and energy efficiency of various workloads and comparison across architectures. The goal of this thesis is to investigate modern power control and measurement technologies available in contemporary processors, and use them to gain knowledge on the efficiency of HPC-relevant workloads.

Contents

1 Introduction **2**

2 Basics **3**

2.1 Hardware . . . 3

2.1.1 Vectorization . . . 3

2.1.2 Caching . . . 5

2.1.3 Dynamic Voltage and Frequency Scaling . . . 6

2.1.4 Energy and Power Measurements . . . 6

2.2 Software . . . 8

2.2.1 Tools . . . 8

3 Experimental Setup **10**

3.1 Hardware Platforms . . . 10

3.2 Benchmarks . . . 11

3.2.1 Heat Stencil . . . 11

3.2.2 Monte Carlo Pi Approximation . . . 11

3.2.3 Stream . . . 12

3.2.4 Vector Multiply-Add . . . 12

3.3 Tools . . . 13

4 Result Analysis **15**

4.1 Heat Stencil . . . 15

4.2 Monte Carlo Pi Approximation . . . 21

4.3 Stream . . . 30

4.4 Vector Multiply-Add . . . 34

5 Conclusion **40**

References **41**

1 Introduction

There is always demand for more performance. Originally the frequency of Central Processing Units (CPUs) was increased to improve computational performance. Subsequently the past 20 years parallelization got popular by adding more cores to CPUs to increase the performance, as well as the addition of Simultaneous Multi-Threading (SMT), through which each CPU core is able to handle multiple threads at once. Another method of parallelization is vectorization where the same operation is applied on whole vectors of data at once without utilizing additional threads. Caches were made larger to load more data and avoid cache misses, which would result in much more frequent memory access to load new data.

With all these technologies and enhancements the transistor budget of CPUs rose and with it the power demand. To exploit SMT some components of CPU cores have to be duplicated and for vectorization there are special execution units which, when utilized, can increase the power consumption of a CPU to such that it has to clock down to stay in its specified limits.

In this work the basics about mentioned concepts and technologies will be explained. The concepts of vectorization and caching will be discussed and how it is possible to read out the energy consumption of a CPU. An interface to read out the consumed energy of CPUs and tools used to manipulate properties of CPUs will be introduced, as well as a script which benchmarks CPUs with different parameters. This thesis will look into the change of energy consumption and wall time as factors of different thread counts and instruction sets on x86 CPUs. Furthermore the automatic power and frequency regulating systems of CPUs and operating systems will be bypassed and it will be analyzed how energy consumption and achieved wall times of a CPU behave on different workloads when limiting the cores' frequency or power consumption.

2 Basics

2.1 Hardware

2.1.1 Vectorization

Through vectorization it is possible to execute the same operation on multiple data elements at once. It is based on a type of parallel processing of Flynn's taxonomy, namely *Single Instruction, Multiple Data* (SIMD). Vectorization often finds use in multi media applications for example where the brightness of a picture should be changed, in which case the brightness of every pixel in the picture is offset by the same value. To execute these operations there is an SIMD execution unit for each CPU core. In figure 1 the SIMD execution unit would take a set of elements (Vector A and Vector B) from two different registers, apply an operation on them and save it in a third register. With usage of larger registers, this can result in a significant speed up but also a higher workload and higher power consumption of the SIMD execution unit.

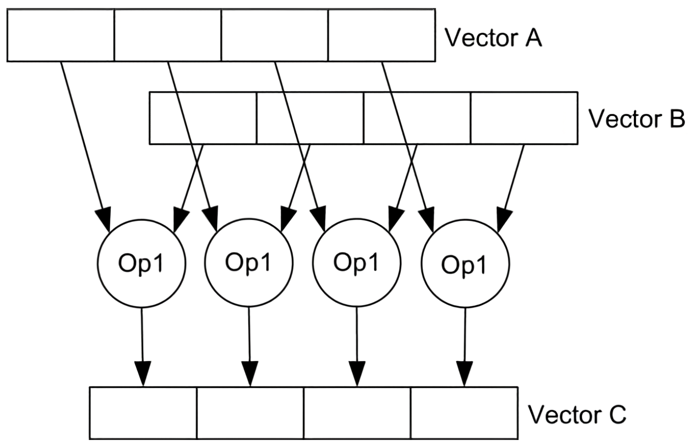


Figure 1: Schema of an operation on the SIMD execution unit [3]

SIMD operations can be exploited through an instruction set architecture such as *Multi Media eXtension* (MMX), *Streaming SIMD Extensions* (SSE) or *Advanced Vector Extensions* (AVX). With MMX Intel introduced a first instruction set for vector operations on integers which operate on eight registers (MM0-MM7). The registers are 64 bit wide, thus one instruction can be applied on one 64 bit integer, two 32 bit integers, four 16 bit integers or eight 8 bit integers.

In this work we will focus on its successor SSE and the more modern AVX instruction sets:

SSE adds 16 separate 128 bit wide SIMD registers (XMM0-XMM15) and instructions for vector operations on single precision floating point elements. Its predecessor MMX shared registers with the x87 floating point unit (FPU) to realize vectorization, because of which it is not possible to mix MMX integer SIMD operations and floating point operations. With wider registers it is possible to load more elements in a register at once (e.g. four 32 bit single precision or two 64 bit double precision floating point elements).

SSE2 is an extension to SSE and adds new instructions for vector operations on double precision floating point elements. It also includes new instructions for integer vector operations to fully replace MMX which still relies on x87 FPU registers for integer vector operations. There are more extensions to SSE (SSE3, SSSE3, SSE4) for other use cases, but these will not be further discussed, since they are not relevant for the topics covered in this thesis.

AVX is an advanced version of SSE. The 16 128 bit XMM registers are extended to 256 bit SIMD registers (YMM0-YMM15, figure 2) and new instructions besides floating point and integer operations got added. It was introduced by Intel with the Sandy Bridge microarchitecture. AVX2 was introduced with Haswell and extended most integer instructions to 256 bit. Since the benchmarks will focus on floating point operations this instruction set is not relevant.

AVX512 doubles the register count to 32 and doubles the size of each register to 512 bit SIMD registers (ZMM0-ZMM31, figure 2).

As compelling as AVX or AVX512 may sound, there is a catch - at least on Intel CPUs. With the introduction of AVX and AVX512, SIMD instructions are so taxing, that the power consumption of the SIMD execution unit can increase significantly. Depending on the instruction a CPU has to perform and how many threads are utilized, the frequency of the CPU may be reduced to stay within its rated power and thermal limits [2], especially for AVX512 workloads. This behavior varies by CPU [2], and does not affect AMD Zen 4 CPUs [11] or Intel Rocket Lake CPUs [5] whatsoever.

The CPUs benchmarked are not affected by this restriction. The only CPU on which AVX512 will be benchmarked is a Zen 4 CPU. One Intel CPU will be limited by power and frequency will not be taken into account. While applying an AVX workload on the rest of the test subjects, there were no changes of frequency observed.

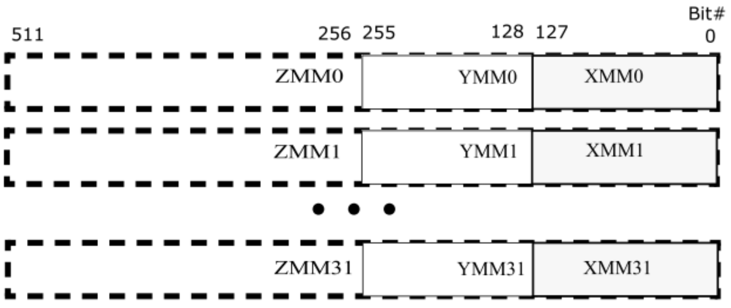


Figure 2: Schema of the registers extensions [12]

All these instruction sets are available on modern x86 architecture CPUs, with the exception of AVX512 which was exclusively available on server grade CPUs in the beginning. Intel supported it for just some generations on desktop models (from 2019 to 2021). AMD supports AVX512 with its Zen 4 microarchitecture (released 2022) for server and desktop models.

2.1.2 Caching

Cache holds data from the random access memory (RAM) needed by the CPU for its computations. In cache, data can be accessed with significantly lower latency and higher bandwidth. The CPU tries to find the needed data first in L1 cache (first level). If it is successful, it is a cache hit, when nothing is found, L2 cache will be accessed and so on. The higher the cache level is, the higher the latency, the lower the bandwidth and the larger the cache. If the needed data is not in the last level cache it is called a last level cache miss and the data will be accessed in the RAM, usually resulting in worse performance except for I/O or network access.

While each CPU core has its own lower level caches, so that each core can have its own data and instruction context, there is usually a higher level cache shared across all CPUs. For x86 CPUs it is common to have core specific L1 and L2 caches and a shared L3 cache. The L1 cache is split up into L1d (data cache) and L1i (instruction cache) which are the same size. But there can be architectural differences for cache. While on Intel's CPUs all cores are contained within one die and with it its L3 cache, AMD realizes its 5000 and 7000 generation Ryzen CPUs with several dies, so called CPU complexes or core complexes (CCX). One CCX contains up to 8 cores each, in case of the AMD Ryzen 9 7900X only 6 cores each. Each CCX has its own L3 cache which is shared across its cores, resulting in two L3 caches shared for a collection of up to 8 cores of a CCX.

2.1.3 Dynamic Voltage and Frequency Scaling

The transistors inside a CPU are switching between their on and off states at the CPU's frequency. For switching the state of a transistor a voltage has to be applied or reduced. The speed at which a transistor switches its state increases with higher applied voltage, thus a higher clock speed of a CPU demands higher voltage to be able to switch the transistors' states properly. The crux is, with increasing the voltage, the power draw increases squared, because by increasing the voltage, current increases as well.

Through *Dynamic Voltage and Frequency Scaling* (DVFS) the CPU automatically regulates its voltage according to its applied workload to run more efficiently. The CPU clocks at higher frequencies when load is applied, and with it the voltage has to be increased to run stably. Vice versa, at lower clocks, lower voltages are sufficient.

In reverse, if DVFS is bypassed by setting a fixed relatively high frequency, the voltage and with it the power draw will increase. Vice versa if we lower the power limit of a CPU, the voltage and with it the frequency will also stay lower.

2.1.4 Energy and Power Measurements

With the Sandy Bridge microarchitecture Intel introduced *Running Average Power Limit* (RAPL), an interface which provides energy readings for different components of a CPU. It also allows to limit the average power of these components. These properties are accessed through multiple power domains [7, Section 14.9.2] (visualized in figure 3):

- Package (PKG): measures the energy consumption of the entire socket, including the processor's core and uncore components
- Power Plane 0 (PP0): measures the energy consumption of the processor's cores
- Power Plane 1 (PP1, client models only): measures the energy consumption of the uncore components of the processor, e.g. integrated graphics processing unit (iGPU)
- DRAM (server models only): measures the energy consumption of the RAM attached to the integrated memory controller

To read out their respective values, RAPL power domains are accessing model specific registers (MSRs). One of which is the MSR_RAPL_POWER_UNIT register which contains the consumed energy in a fictional unit as an unsigned integer by bits 12:8 called *Energy Status Units* (ESU) (as in figure 4). The estimated energy consumption in Joules is based on the multiplier $1/2^{ESU}$. Each power domain has its own MSR_RAPL_POWER_UNIT register and with it separate ESU value. This MSR is 32 bit wide and can overflow when enough energy was consumed [9]. To detect these overflows and prevent wrong results, the counter gets polled. [1]

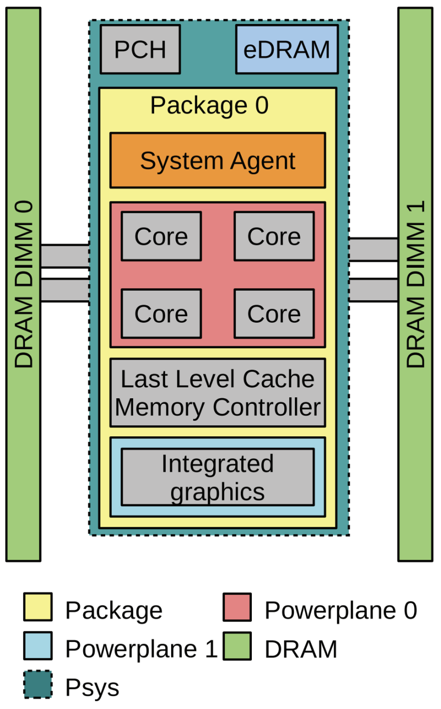


Figure 3: Schema of the RAPL power domains [9]

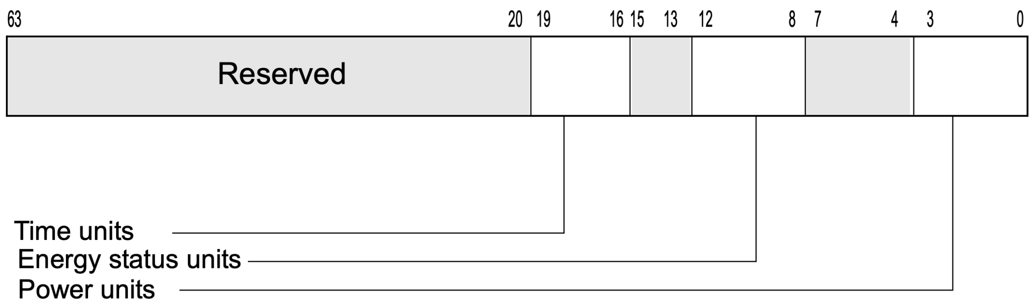


Figure 4: MSR_RAPL_POWER_UNIT register [7]

There are different MSRs for different microarchitectures (hence the name). Fortunately the MSR_RAPL_POWER_UNIT can be accessed (e.g. through `perf`) the same for Intel microarchitectures since Sandy Bridge. AMD CPUs on the other hand are compatible with RAPL since the Zen 2 microarchitecture, but the MSRs do not correlate one to one with Intel's microarchitectures. For instance the package domain can still be accessed, but no specific power plane domains, which are located in other MSRs not as easily accessible through `perf`.

On non-x86 architectures like ARM there do exist similar registers but those architectures are not compatible with RAPL, their registers can not be accessed through RAPL.

Thus the focus will be on Intel and AMD microarchitectures only.

2.2 Software

2.2.1 Tools

In order to accurately measure the results, several tools were used. These tools facilitated the realization and setting the parameters of the benchmarks, as well as measuring the energies consumed and the time passed while running a benchmark.

perf is a performance analysis tool which obtains information about the CPU by reading out specific registers. In this case it is used to read out the energy data through RAPL.

Before and after each run of a benchmark the value of ESU of the MSR_RAPL_POWER_UNIT register can be read out to evaluate the approximately consumed energy of the CPU during the execution of the benchmark. Wall time can also be measured with `perf`. Energy data readouts are realized with `perf stat`. An example call – where the energy consumption of the PP0 domain and wall time are tracked for the execution of a benchmark – would look like this:

```
> perf stat -e power/energy-cores/ ./benchmark_executable
```

The command returns the following output:

```
# started on Mon Nov 7 18:40:47 2022
 Performance counter stats for 'system wide':
             10.63 Joules power/energy-cores/
       0.771935730 seconds time elapsed
```

In this command `energy-cores` refers to the PP0 power domain. `energy-pkg` would refer to the PKG power domain.

cpupower is a command line interface (CLI) tool which allows to get CPU specific information and scale the voltage and the frequency of CPUs manually and bypass DVFS. Here it is used to set the CPU's scaling governor and limit its frequency. Scaling governors specify at which frequency the CPU runs, and can be seen as profiles which are optimized for different use cases [4]:

- *performance*: maximum specified frequency

- *powersave*: minimum specified frequency
- *ondemand*: dynamic scaling, maximum frequency under load, minimum frequency in idle
- *userspace*: user specified frequency

By setting the scaling governor to *userspace* the frequency can be set freely to a certain degree. On the Core i7 3770 it is possible to set the frequency in 100 MHz steps between the determined minimum and maximum frequency. On AMD CPUs however only some predefined frequencies are available (e.g. the Ryzen 7 5800X provides only 2.2 GHz, 2.8 GHz and 3.8 GHz as valid values).

powercap-set allows to set power limits for CPUs through the Linux power capping framework. Power limits can be set for different CPUs, if multiple are available, and limits can be set for the power domains of each CPU. Here they are called power zones and subzones. For instance: *intel-rapl:0:0* (power subzone 0 of CPU 0) corresponds to the core power domain (PP0) of CPU 0 of a system. *intel-rapl:0:1* would be the subzone for the uncore power domain (PP1).

Furthermore limits can be set for different constraints, creatively named *constraint-0* and *constraint-1*. Constraints correlate to different time windows over which a defined power limit is applied. *constraint-0* being the short term and *constraint-1* being the long term constraint. Power limits and time windows can be set separately for each constraint. Constraint specific limits are applicable for each zone and subzone separately. [13]

compiler flags: by adding the option `-march=native` all instruction subsets available for the local machine are enabled [6] for the compiler. The compiled program may not run on other systems - which possibly do not support the same instruction sets - but optimized it for the CPU of the system the program was compiled. For example when using AVX intrinsics, by adding `-march=native`, the required `-mavx` is implicitly included, which allows the compiler to use AVX instructions. Besides that, different optimization flags will be used as well as `-fopenmp` in order to access usage of OpenMP pragmas.

OpenMP is an API which provides pragma-based parallelization. OpenMP pragmas are used here to parallelize compute intensive benchmarks like the pi approximation as well as the stream memory benchmark. OpenMP would also provide pragmas to realize vectorization, but since it is not possible to define through an OpenMP pragma which instruction set should be used to exploit vectorization, intrinsics from the corresponding instruction set will be used.

3 Experimental Setup

3.1 Hardware Platforms

To look at more aspects of CPUs, several CPUs with slightly different feature sets will be benchmarked.

The Intel Core i7 3770 is a desktop client CPU based on Intel's Ivy Bridge microarchitecture, released in 2012. It supports the SIMD instruction sets SSE, SSE2 and AVX, its frequency can be set via `cpupower` between 1.6 GHz and 4.1 GHz in 100 MHz increments and provides 4 physical cores, with SMT 8 logical cores respectively. Each core possesses 32 kiB L1 cache and 256 kiB L2 cache and shares 8 MiB L3 cache. Unfortunately it is not possible to set power limits for this CPU through `powercap-set`, or at least not on the tested system specifically.

Intel's Xeon E5-4650 server grade CPU also released in 2012 but still is based on the Sandy Bridge microarchitecture. Among others it supports the same three mentioned instruction sets and has the same amount of cores as its desktop counterpart, but has a larger L3 cache with 20 MiB. Also the frequency is not as high, with up to 2.7 GHz. The system tested features 4 of these CPUs in one cluster. Fortunately on this system it is possible to limit the power draw. However, it is not possible to set a fixed frequency.

In 2020 AMD released its Ryzen 7 5800X with 8 physical and 16 logical cores (contained in one single CCX) based on the Zen 3 microarchitecture. Just as the i7 3770 it is a desktop CPU and supports SSE, SSE2 and AVX. L1 cache size is the same as for the Intel models at 32 kiB, L2 cache is doubled to 512 kiB per core and the L3 cache size is 32 MiB shared across all cores. Being released 8 years after the other two CPUs it is expected to be faster and more efficient, thus needing less energy, even if its TDP and thus its power draw is significantly higher than for the i7. This CPU does not support power draw limiting, its frequency can be set manually, but as already mentioned, to only three fixed values: 2.2 GHz, 2.8 GHz and 3.8 GHz.

Two years later AMD released the Ryzen 9 7900X with 12 physical and 24 logical cores (distributed over two CCXs) based on the Zen 4 microarchitecture. Additionally to the other SIMD instruction sets this CPU also supports AVX512. For this CPU again L1 cache size stays the same and the L2 cache is doubled to 1 MiB per core, L3 cache size is 32 MiB per CCX. The frequency can be set to 3.0 GHz and 4.7 GHz.

On Intel systems the energy consumption of the PP0 power domain will be measured, since PP0 only contains the cores. Therefore the energy consumption of other devices on the CPU will be ignored. On AMD systems it is not possible to access the PP0 domain (at least not as trivially), so the power consumption of the PKG domain will be measured.

This can lead to some noise in the measurements, because the energy consumed by the memory controller in the IO chip will also be measured. Nevertheless there is no iGPU in the package. With its next generation of CPUs, AMD added an iGPU, such as the R9 7900X tested, which will potentially make even more noise in the energy measurements of this CPU, time measurements should not be affected by this.

3.2 Benchmarks

In order to analyze the energy consumption behavior of different aspects of a CPU, multiple benchmarks are set up to exploit these aspects. There will be no thread pinning. Threads will be assigned by the OS's scheduler. The benchmarks are written in C which allows usage of OpenMP and intrinsics. The source code of the benchmarks can be accessed via GitHub [10].

3.2.1 Heat Stencil

This benchmark simulates the propagation of heat in a two dimensional grid. In this grid each cell has a value such as temperature and is updated accordingly based on neighboring segments, simulating heat dissipation. The benchmark is parallelized using multi threading. The heat dissipation occurs in iterations of time steps. In order to process the next iteration, after each time step, each thread needs to update the state of the temperature of the gird cells it computed for the other threads. This has to be loaded into a shared cache layer. Thus cache performance can have impact on the result.

3.2.2 Monte Carlo Pi Approximation

Monte Carlo pi approximation is a numeric way to approximate the value of pi and is known as a embarrassingly parallelizable computation. This benchmark will be used to analyze the systems being computationally bound.

Pi is approximated by analyzing if randomly placed points are inside a circle or not. Each thread analyzes its own set of points and computes its own approximation of pi. Since this approximation is based on the law of large numbers, using more points yields a more accurate result, thus taking the arithmetic mean of each of the thread yields more accurate result as well. The formula for the approximation is this:

$$\pi \cong \frac{points\ inside\ quadrant}{all\ points} * 4$$

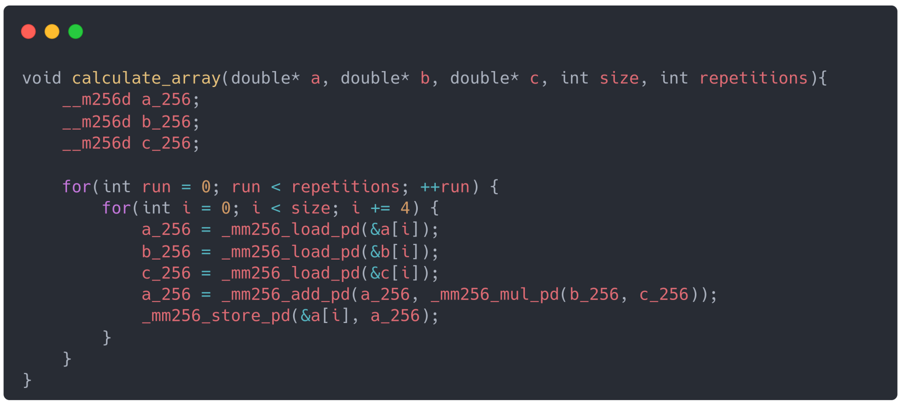
3.2.3 Stream

The Stream benchmark is a common standard benchmark for measuring sustained memory bandwidth [8]. In this benchmark four operations are benchmarked on arrays of different sizes. Array sizes will range from several kB to fit into the L1 cache up to multiple GB to provoke memory accesses. With smaller arrays the tested system will be compute bound, which means the computational performance of the CPU will be the bottleneck. With larger array sizes the system will be memory bound and will permanently access memory to get all the needed data and will be bottlenecked by the memory bandwidth at which RAM will be accessed. It will be analyzed how bandwidth scales with frequency and energy consumption.

3.2.4 Vector Multiply-Add

In this benchmark two vectors are element wise multiplied and added onto another vector. It will be used to analyze the advantages of using vectorization.

Vectorization will be realized here with intrinsics. Intrinsics can be seen as one level above writing assembly instructions. Through intrinsics one can tell the compiler to use specific instructions, in this case to use instructions from specific SIMD instruction sets.



```
void calculate_array(double* a, double* b, double* c, int size, int repetitions){
    __m256d a_256;
    __m256d b_256;
    __m256d c_256;

    for(int run = 0; run < repetitions; ++run) {
        for(int i = 0; i < size; i += 4) {
            a_256 = _mm256_load_pd(&a[i]);
            b_256 = _mm256_load_pd(&b[i]);
            c_256 = _mm256_load_pd(&c[i]);
            a_256 = _mm256_add_pd(a_256, _mm256_mul_pd(b_256, c_256));
            _mm256_store_pd(&a[i], a_256);
        }
    }
}
```

Figure 5: Vector Multiply-Add with AVX intrinsics for double precision

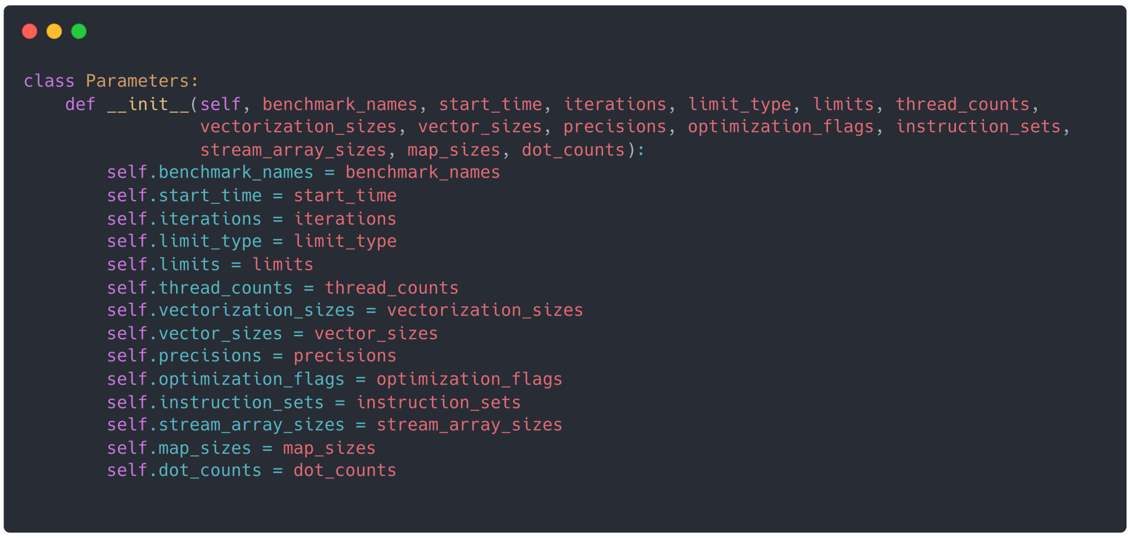
In figure 5 AVX intrinsics for double precision floating point elements are used. With the `_mm256_load_pd()` calls four elements at a time are loaded from the arrays `a`, `b` and `c` in the memory into the YMM registers `a_256`, `b_256` `c_256`. Since YMM registers are 256 bit wide, they can contain four 64 bit double precision floating point elements, so that with every iteration of the loop, four indexes can be skipped. The `_mm256_mul_pd()` intrinsic takes two registers as arguments and multiplies them component wise. The `_mm256_add_pd()` intrinsic adds two vectors component wise. The result loaded in `a_256` is finally stored again into the memory in the array `a` in order to be loaded again in the next iteration.

3.3 Tools

The benchmarks are executed automatically with a python script on the target system. The benchmarks are called through perf, the output of which - the consumed energy and the wall time - is saved into a text file for each benchmark run. The files are saved in separate folders for each benchmark. The command is called in the script through `subprocess.run()`. The script waits for the subprocess to be finished before it continues with the next command or the next iteration. While waiting for the subprocess there is no measureable increase in energy consumption.

While the parameters relevant for Monte Carlo pi approximation and the heat stencil benchmark can be applied by passing them as arguments at the call of the executable, the stream and vector multiply-add benchmarks have to be complied to apply certain parameters. To use the stream benchmark as intended, the thread count is set by the environment variable `OMP_NUM_THREADS`, which has to be set before compilation, the array size is passed as a constant with `-DSTREAM_ARRAY_SIZE=array_size` at compilation. Scaling governor and frequencies are set automatically through `cpupower`, power limits as well through `powercap-set`. Power limits are set to the same value for both constraints.

To start the benchmarks the parameter space has to be defined with a Parameters object.



```python
class Parameters:
    def __init__(self, benchmark_names, start_time, iterations, limit_type, limits, thread_counts,
                 vectorization_sizes, vector_sizes, precisions, optimization_flags, instruction_sets,
                 stream_array_sizes, map_sizes, dot_counts):
        self.benchmark_names = benchmark_names
        self.start_time = start_time
        self.iterations = iterations
        self.limit_type = limit_type
        self.limits = limits
        self.thread_counts = thread_counts
        self.vectorization_sizes = vectorization_sizes
        self.vector_sizes = vector_sizes
        self.precisions = precisions
        self.optimization_flags = optimization_flags
        self.instruction_sets = instruction_sets
        self.stream_array_sizes = stream_array_sizes
        self.map_sizes = map_sizes
        self.dot_counts = dot_counts
```

Figure 6: Parameters class

Each list of values of a parameter is looped through for a benchmark. To mitigate outliers each run is executed for multiple iterations. In the evaluation the arithmetic mean of all iterations of all runs with the same parameter will be formed. The parameters set for the benchmark are saved in a config file for easier plotting of the data.

4 Result Analysis

The results discussed in this section can be accessed via the GitHub repository associated to this thesis. [10]

4.1 Heat Stencil

The heat stencil benchmark is compiled with GCC 12 using `-O2` – which achieved better results regarding energy consumption – and `-march=native`. Parameters were set to `thread_count=4` – except for comparisons between different thread counts – and `map_size=400` resulting in a 2D grid of 400x400 cells. The simulated temperature is stored with double precision. The grid is initialized with the same temperature for every cell except one, which has a higher temperature and from which heat will be dissipated.

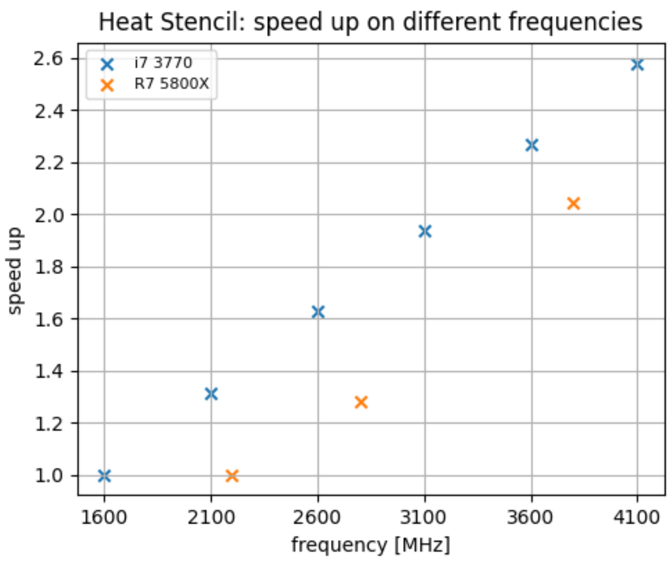


Figure 7: Relative speed up of Core i7 3770 and Ryzen 7 5800X to their lowest settable frequency

As expected with higher frequency the speed up scales linearly for the i7 3770 as seen in figure 7. The more modern R7 5800X is significantly faster than the i7, even when running at similar frequencies (i7 at 2.1 GHz: 2.04 seconds; R7 at 2.2 GHz: 0.75 seconds). This can have enhancements of a more modern microarchitecture as a cause, for example a higher IPC (instructions per cycle) which allows a CPU to execute more operations in the same time at the same frequency - or in other words per cycle - than CPUs based on older microarchitectures. This can be achieved by, for example, better branch prediction, lower latency on cache, larger registers or more execution units, which again results in more needed die size and higher power power draw. Also worth noting is that when increasing the frequency of the R7 by 27% from 2.2 GHz to 2.8 GHz the speed up is about 1.27, but increasing the frequency by 73% to 3.8 GHz results in a speed up of over 2, which seems odd. After further investigation, limiting the frequency of the AMD CPU to 3.8 GHz, it only mostly stayed at that frequency, sometimes it just ran at its fastest possible frequency above 4 GHz (maximum frequency is adjusted adaptivly), which explains the superlinear speed up. This behavior could not be observed on the other systems.

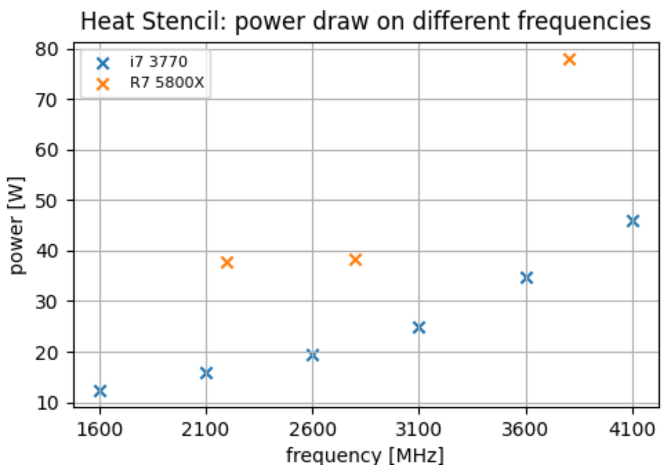


Figure 8: Power draw of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

With measuring energy and time the average power draw of the CPUs can be calculated. In figure 8 there is a much higher power draw visible for the R7 5800X. This is likely due to the aforementioned improvements. For instance more execution units logically do draw more power.

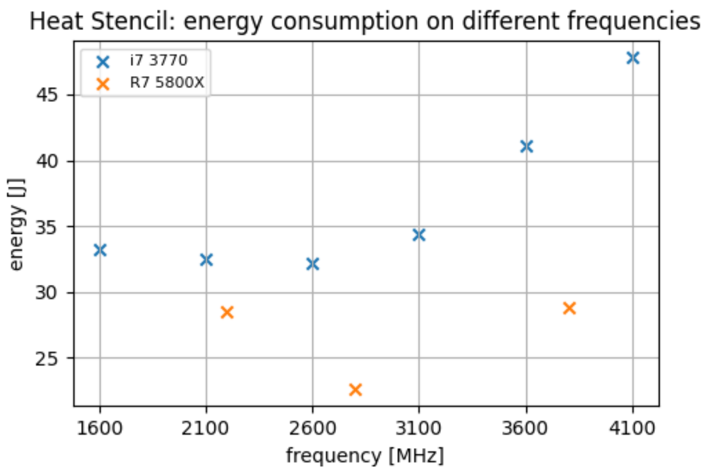


Figure 9: Energy consumption of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

With higher frequency the power draw rises superlinearly, but the speed up just linearly. As a consequence there exists a sweet spot for frequency in regard to energy efficiency (seen in figure 9), which is at about 2.2 GHz for the Intel CPU and at about 2.8 GHz for the AMD CPU. It could be that there is a more energy efficient frequency for the R7, but due to a lack of available frequencies to set, a better picture is not attainable. It is worth mentioning that the difference of consumed energy for frequencies below the sweet spot is way higher for the R7 than for the i7, since power draw rises only measurably at 2.8 GHz for the R7. When running at 2.8 GHz only about 22.5 Joules are consumed compared to about 28 Joules at 2.2 GHz. Furthermore the consumed energy is about the same for the R7 regardless of running it at 2.2 GHz or 3.8 GHz while the wall time is halved from 750 milliseconds to about 370 milliseconds when running with the higher frequency. Moving below the frequency sweet spot for the i7 energy consumption just rises slightly with lower frequency, however, moving above the sweet spot energy consumption rises roughly linearly.

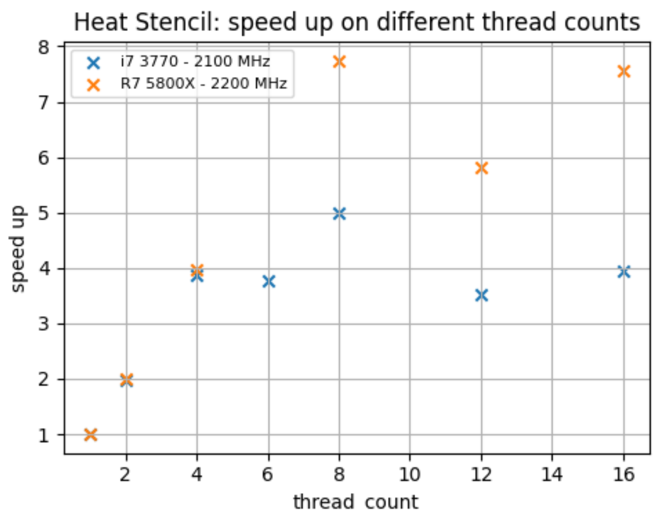


Figure 10: Wall time of heat stencil benchmark of Core i7 3770 and Ryzen 7 5800X regarding thread count

When increasing the thread count the wall time scales nearly perfectly until 4 threads for the i7 and 8 threads for the R7, their respective physical core count. Increasing the thread count further also increases the wall time slightly in the beginning but decreases again with higher thread count. This is an instance of oversubscription of threads. For the i7 when running the benchmark with 4 threads, each thread is assigned to their own physical core and will likely not be rescheduled. When trying to assign 6 threads, the two remaining threads will be scheduled to less utilized cores. Which means they will be rescheduled on a frequent basis since the core they are assigned to is more utilized, because it has to handle two threads at once. On rescheduling, the cores have to perform a context switch which is relatively time expensive. Assigning a thread count which equals the amount of logical CPU cores the wall time stays the same or decreases slightly due to potentially better utilization of physical cores. The same applies for the R7.

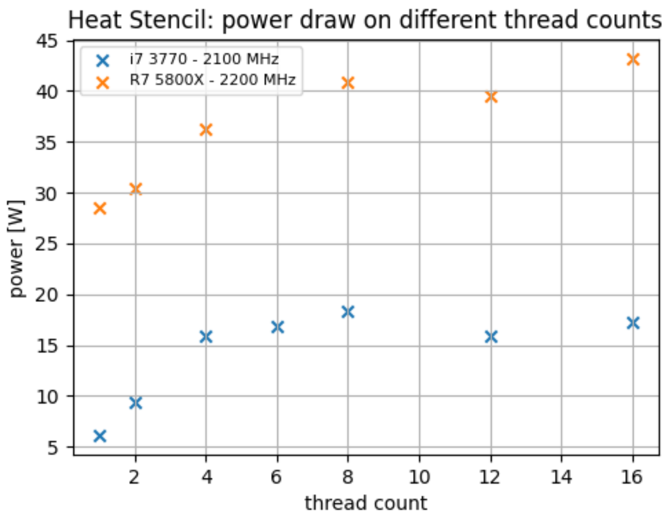


Figure 11: Power consumption of heat stencil benchmark of Core i7 3770 and Ryzen 7 5800X regarding thread count

Power draw scales linearly with thread count for the i7 until 4 threads. As seen with the wall times, the physical cores are already well utilized with 4 threads, thus assigning more threads results in just slightly higher power demand. When assigning 8 threads less time is wasted for rescheduling and the cores are better utilized. Looking at the AMD system there is sublinear scaling with thread count. While the power draw of the cores themselves may scale linearly, the power draw of the IO chip of AMD CPUs may not with higher thread counts.

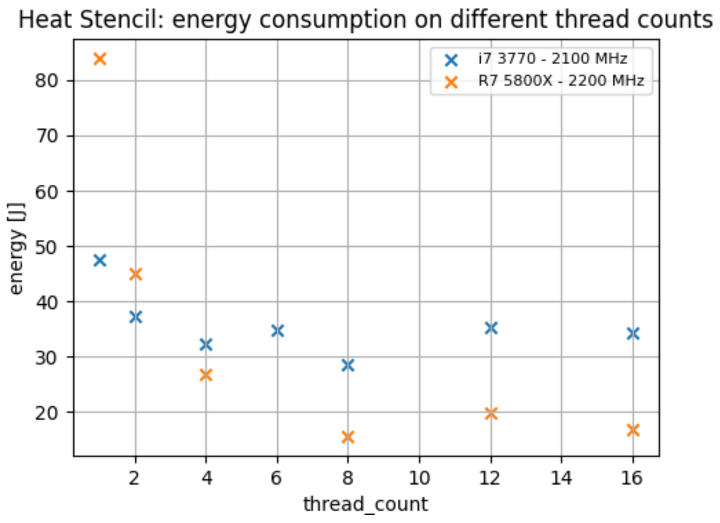


Figure 12: Energy consumption of Core i7 3770 and Ryzen 7 5800X regarding thread count

Again, because of not measuring just PP0 for the AMD system, the measured energy consumption is higher. But using more cores or threads results in higher energy efficiency. When assigning for example just one thread, only one core will be utilized, while the other cores will stay in idle, consuming less energy as the utilized core. But even being idle they draw power which does not contribute to faster computation but to a higher energy consumption because of the higher wall times for executing the benchmark with just one thread. Same as with wall times, there is a small increase of energy consumption, because of oversubscription on threads. Again this effect lessens when approaching the count of logical available cores.

Using different compiler optimization flags for the benchmark did not show much fluctuation in power consumption but significant fluctuation in wall time and also in energy consumption. The major take away here is using `-O2`, `-O3` and `-Ofast` results in a significant speed up and thus lower energy consumption (figure 13 and figure 14) due to lower wall times. Not optimizing – using `-O0` – and using `-Os` should be avoided in this regard, while it is worth noting that using `-Os` can result in more efficient disk usage, since it optimizes for an executable with a smaller file size. However, as an edge case this can be advantageous by a more efficient usage of the L1 instruction cache, but this will not be further investigated.

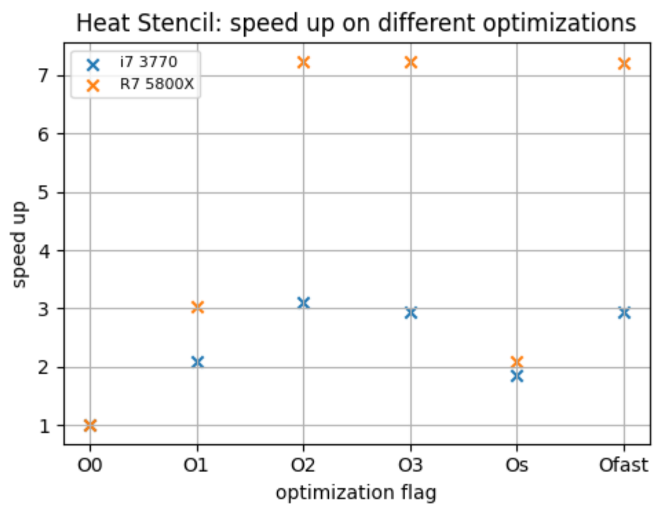


Figure 13: Speed up of Core i7 3770 and Ryzen 7 5800X regarding compiler optimization flags

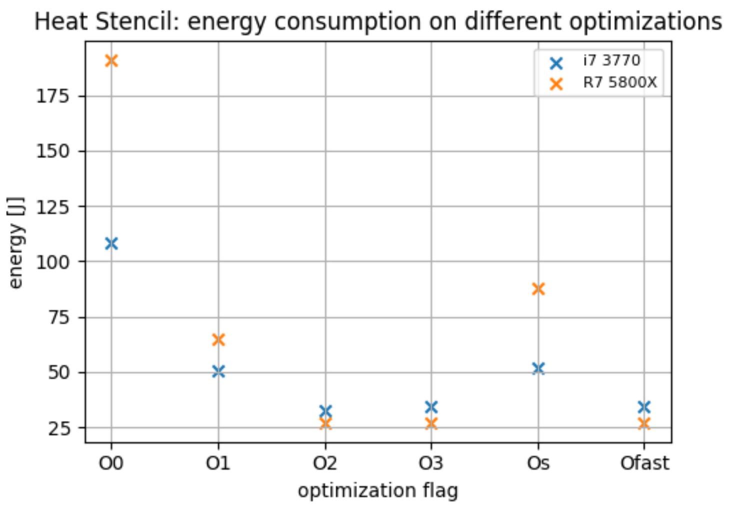


Figure 14: Energy consumption of Core i7 3770 and Ryzen 7 5800X regarding compiler optimization flag

4.2 Monte Carlo Pi Approximation

For Monte Carlo pi approximation (`dot_count=640000000`, `thread_count=4`, compiled with GCC 12 and `-O2` optimized) the results look similar to the heat stencil benchmark in regards to frequency. While speed up (figure 15) and power consumption (figure 17) scales the same as with heat stencil benchmark, the energy consumption is way higher (figure 16).

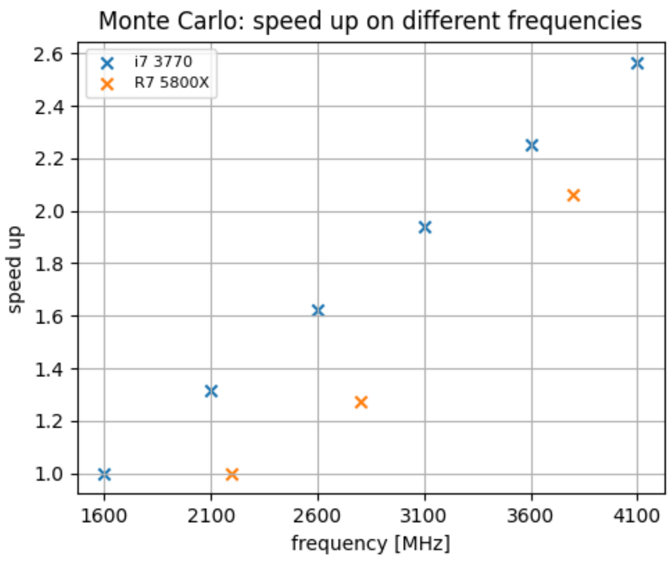


Figure 15: Speed up of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

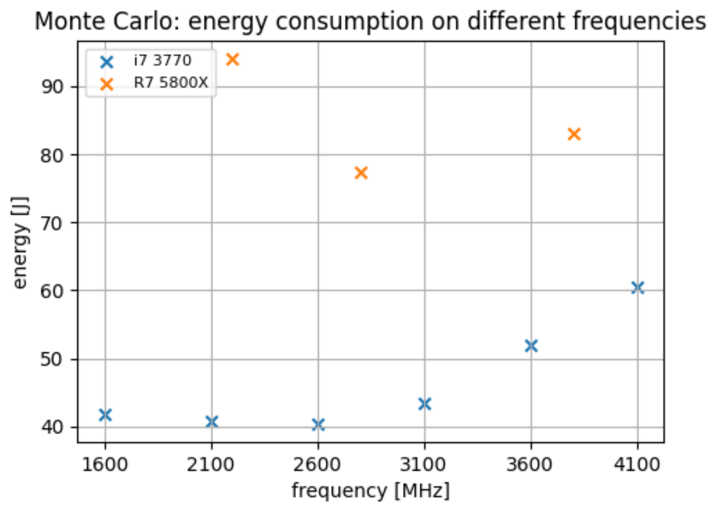


Figure 16: Energy consumption of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

This is due to the higher power consumption of the R7 (figure 17) while achieving roughly the same wall times as the i7 (figure 18).

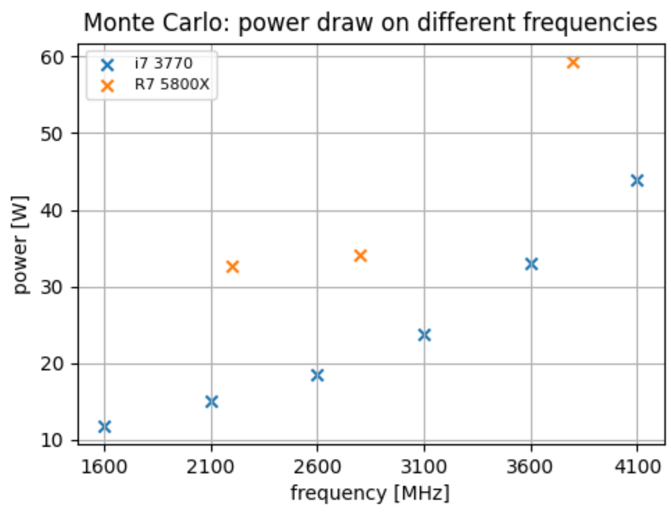


Figure 17: Energy consumption of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

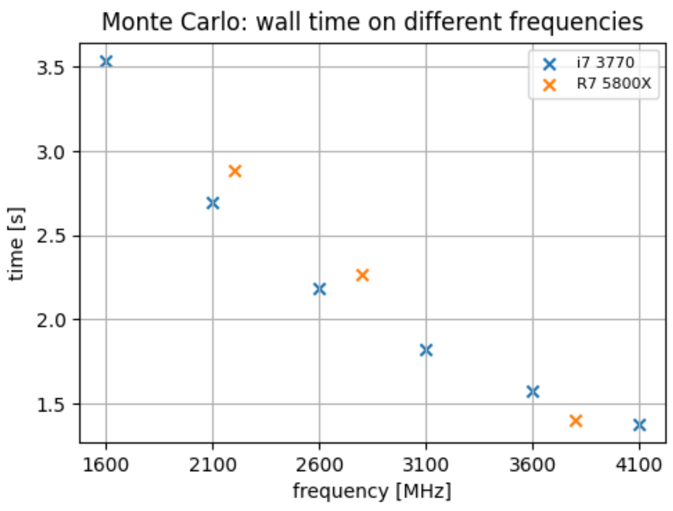


Figure 18: Wall times of Core i7 3770 and Ryzen 7 5800X regarding frequency on 4 threads

Looking at the thread count yields more deviating results. For both systems the oversubscription of threads is not as bad as with heat stencil, but there is still a slightly lower scaling of speed up, when using SMT. When assigning more threads than available the speed up does not change significantly for the i7.

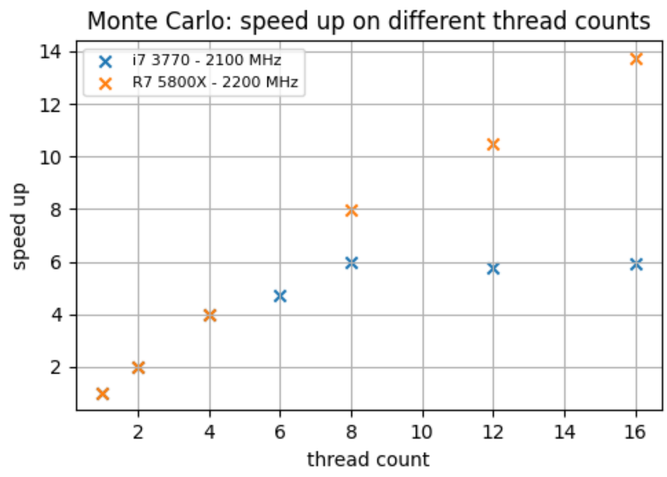


Figure 19: Speed up with different thread counts

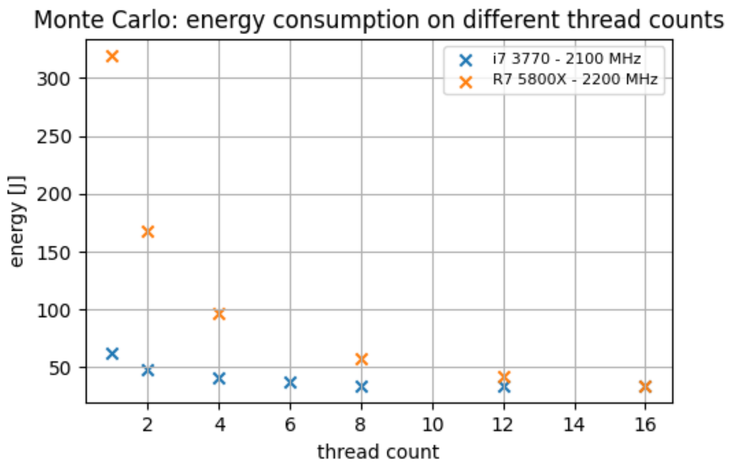


Figure 20: Energy consumption with different thread counts

Also optimizing the benchmark while compilation yields different results. Regardless of the optimization, the speed up stays the same (figure 21), thus reduction of energy consumption also is the same (figure 22). Optimization is way easier for the pi approximation than for heat stencil, since it is a way more trivial brute force algorithm and does not rely as much on cache as the heat stencil benchmark.

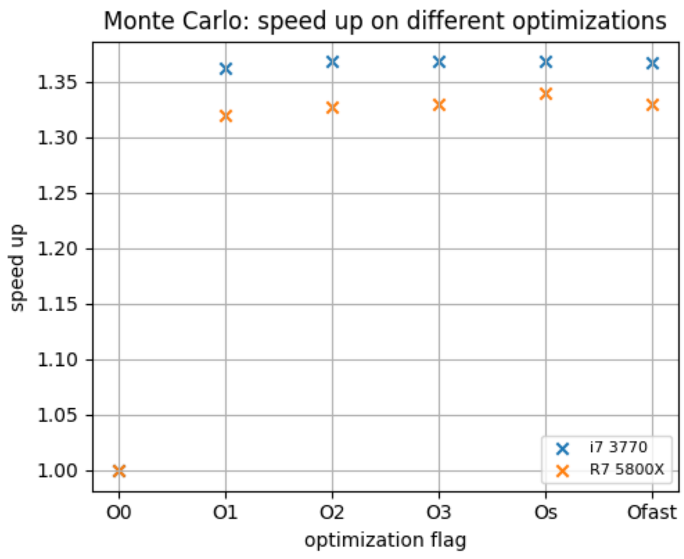


Figure 21: Speed up with different optimizations

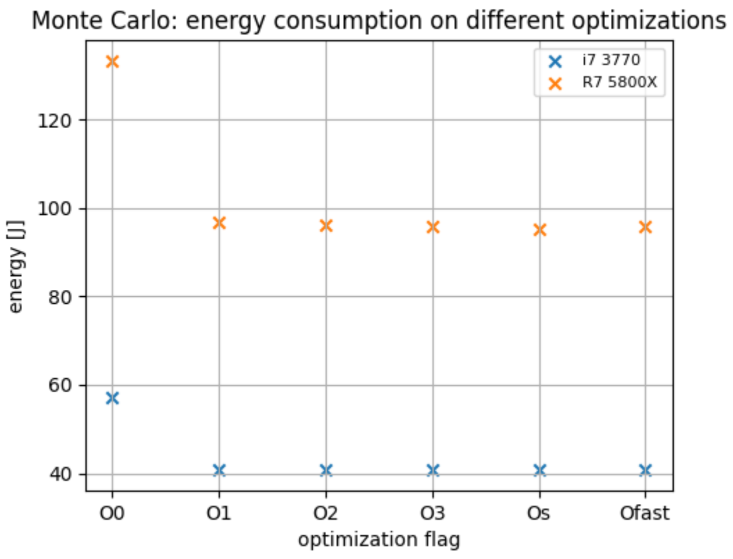


Figure 22: Energy consumption with different optimizations

Looking at the data of the Xeon E5-4650 where power draw is set to different limits (maximum power draw is 130 W), there are also unexpected results. When setting the power limit in 20% increments, speed up should have a sublinear scaling, since — as observed — there is a superlinear scaling of power draw with frequency. But speed up does not scale as expected above power limits of 52 W (figure 23).

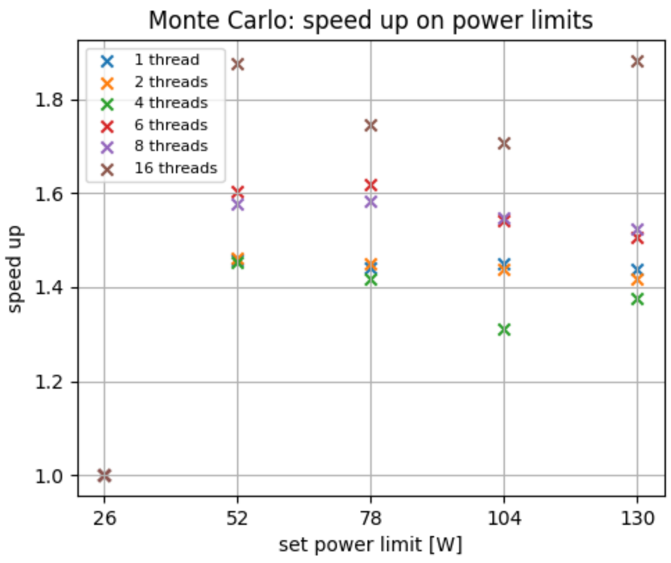


Figure 23: Speed up with different power limits and thread counts

Because there is a speed up when setting the power limit higher than 26 W, the CPU runs at higher frequencies, likely at a rather high one already, since the speed up does not increase with power limits higher than 52 W. With higher frequencies and thus with higher power limits there should be a higher energy consumption, nevertheless it does not change significantly as seen in figure 24, except for 26 W, where the CPU runs at a lower and more energy efficient frequency. Again with power limits of 52 W and above, the CPU likely runs at a frequency at or near its upper limit, because the energy consumption does not change with higher power limits.

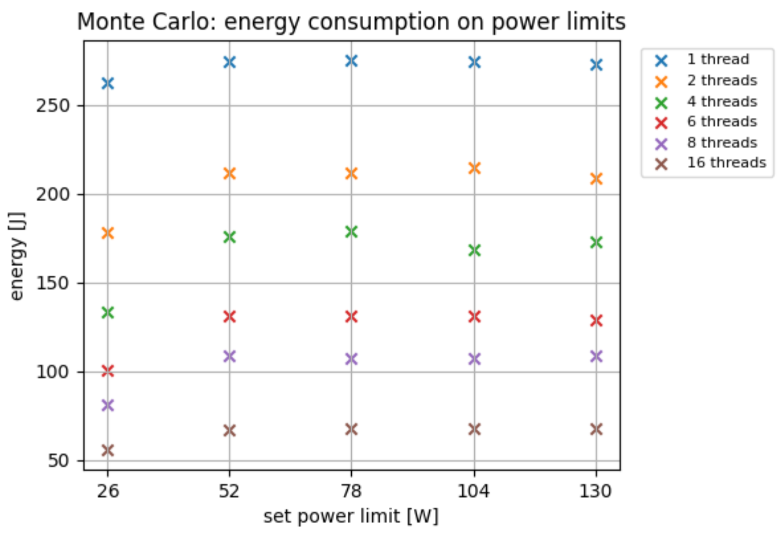


Figure 24: Energy draw with different power limits and thread counts

Calculating the power draw by time and energy measured, the power draw is way higher than the set power limits (figure 25). The power limiting through `powercap-set` does have an affect on lower power limits but does not limit power as strictly as expected. One could assume that a limit of 26 W is applied per core as there is a power draw of about 26 W with only one thread and about 52 W with two threads, but the power limit is applied to the whole PP0 power domain, so utilizing more cores should not result in higher power draw.

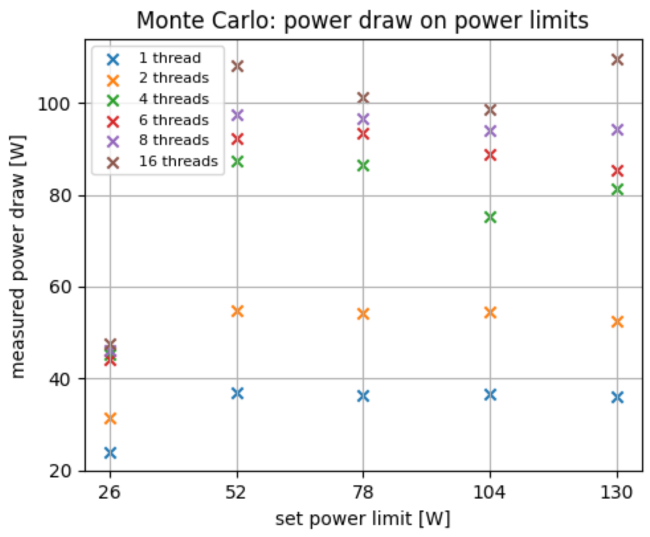


Figure 25: Power draw with different power limits and thread counts

Setting the power limit in 5% increments up to 25% (39 W) results in the following power draw seen in figure 26. When utilizing 4 threads there is a somewhat reasonable scaling visible but the power draw is still higher than the set limit. The difference between the different power limits is also a bit too high, with 6.5 W increments, the power draw increases by about 6.5 W on lower limits but also increases by about 10 W at higher power limits. For all other thread counts the scaling is sometimes more reasonable, sometimes less. Especially at set power limit of 13 W there is a rather low measured power draw for all thread counts except 4. For 1 or 2 threads the power draw does stagnate at measured 37 W and 45-68 W respectively, since on lower thread counts the assigned cores can not draw more power.

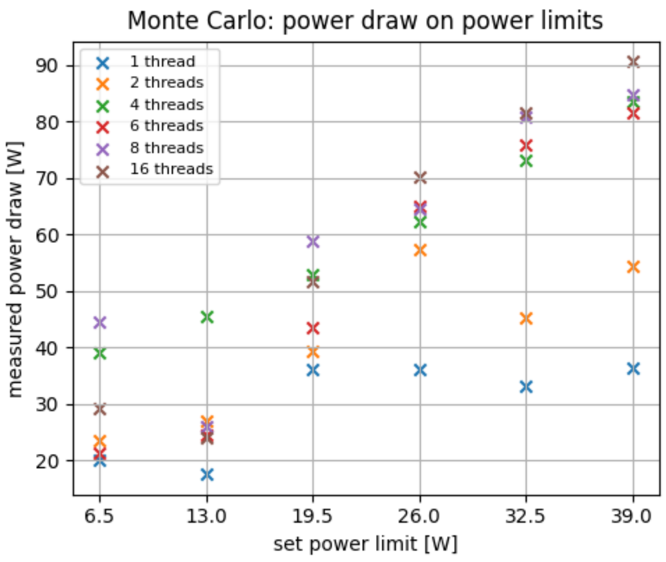


Figure 26: Energy consumption with different power limits and thread counts

In figure 27 for thread counts of 4 and 8 the scaling of speed up is expected, but for way lower set power limits. The respectively measured power consumption does fit to the measured speed up for these thread counts, since there is a flattening of the speed up with higher power draw. The speed up for the other thread counts – especially 6 threads – is unrealistically high. At the power limit of 13 W the speed up is lower than 1 for some thread counts, which correlates with the measured power consumption for this power limit in figure 26. The flattening of the curve with higher set power limits indicates, that the CPU would be near its power limit. This correlates with the data shown in figure 25 where the power drawn from the PP0 power domain is about 80 to 90 W at a limit of 39 W. The TDP of the Xeon is at 130 W which includes the rated power draw for the whole package and not just PP0.

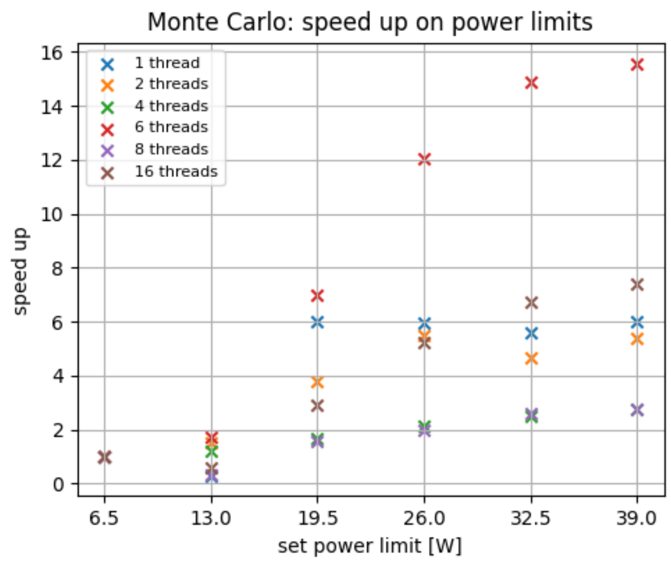


Figure 27: Speed up with different power limits and thread counts

At a power limit of 13 W there is an unexpected high energy consumption when using 1 thread. The trend of lower energy consumption with higher thread count is visible again in figure 29. The energy consumption does not decrease much with higher power limits.

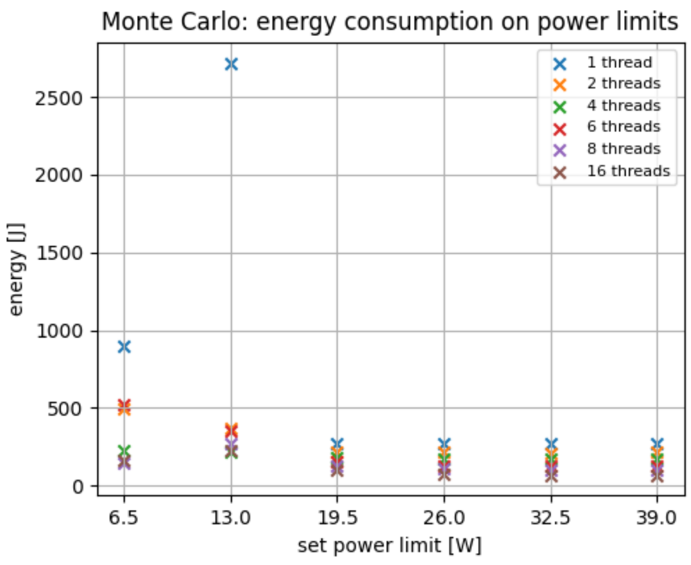


Figure 28: Energy consumption with different power limits and thread counts

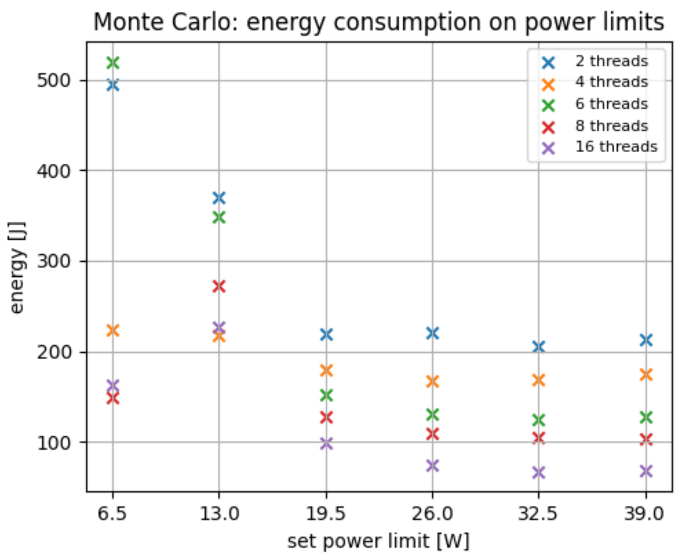


Figure 29: Energy consumption with different power limits and thread counts excluding data for 1 thread

Because the Xeon is the only CPU on which `powercap-set` achieved any impact, the phenomenons viewed in this section could not be reproduced on the other systems.

4.3 Stream

Of the result from the stream benchmark the bandwidths of the copy operation will be analyzed. The benchmark was compiled via the Makefile provided from the GitHub repository [8]. Compiler flags included in the Makefile are `-O2` and `-fopenmp`. The data shown is based on parameters of `thread_count=4` and `precision=double`. The array sizes shown in the figures are determined by the element count of the array multiplied with the size of a double precision floating point element in bytes.

With smaller array sizes the bandwidth is dependent on the frequency of the CPU, which represents a compute bound case where the array is small enough to fit in the cache. When increasing the frequency, the bandwidth scales linearly. With larger array sizes the system is memory bound and the bandwidth drops to a fraction of what it is when computationally bound. Data does not fit in cache and must be loaded from RAM, which does not achieve such high bandwidths. While the bandwidth is reduced by a factor of 5 at 4.1 GHz from a bit over 50 GB/s to a bit over 10 GB/s, it does not even drop by half at 2.2 GHz since at that frequency the bandwidth is not that high to start with. For the array size of 3.2 MiB (the third set of points from the left in figure 30) the bandwidth drops just for higher frequencies at which memory boundness starts to take effect.

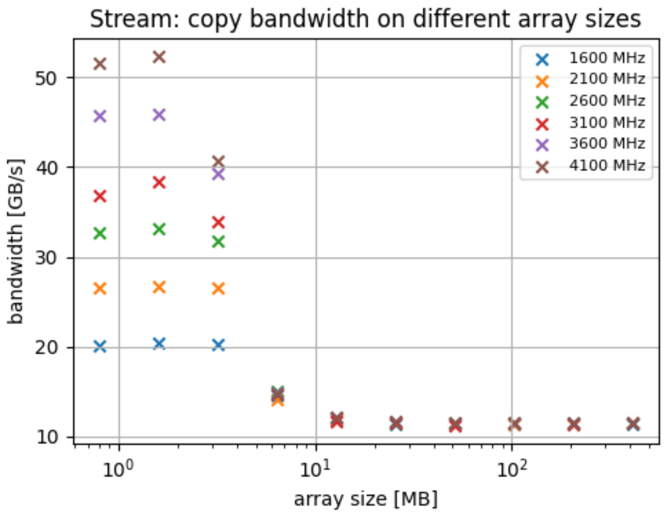


Figure 30: Bandwidths of i7 3770 with different array sizes

The AMD system shows the same effect of memory boundness with larger array sizes. The Intel system starts to drop bandwidth at an array size of 3.2 MiB whereas the AMD system shows this effect at around 12.8 MiB. This is likely due to the four times larger L3 cache (8 MiB vs. 32 MiB) of the AMD CPU. The bandwidth reaches up to about 150 GB/s when compute bound and drops down to under 20 GB/s when memory bound. While bandwidth scales linearly with frequency for the Intel system, on the AMD system it scales more than just linearly. For an array size of 3.2 MiB increasing the frequency from 2.2 GHz to 2.8 GHz (27%) the bandwidth increases about 67 GB/s to 98 GB/s (46%). When increasing the frequency to 3.8 GHz (73%) bandwidth even increases by 137% to almost 160 GB/s.

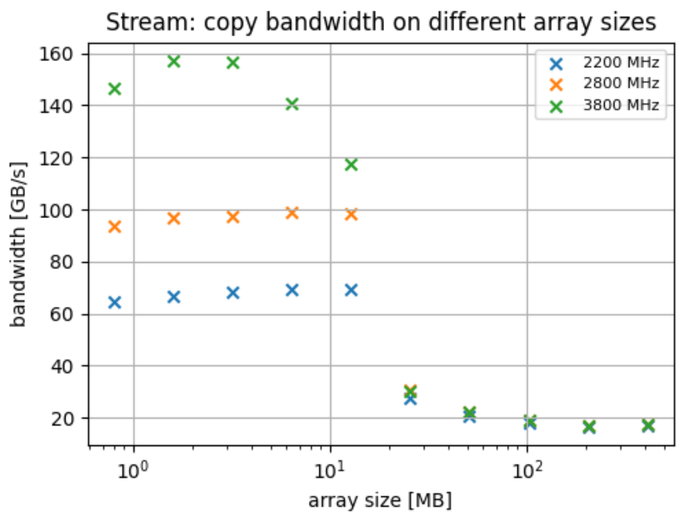


Figure 31: Bandwidths of R7 5800X with different array sizes

In the compute bound case (1.6 MiB) there is a similar scaling of speed up and energy consumption with frequency as seen before at the compute intensive heat stencil benchmark. The same can be said for scaling of energy consumption with frequency.

However, for the memory bound case (51.2 MiB) it looks worse. With higher frequency there is speed up but only by a small margin (figure 32) whereas the energy consumption rises regardless (figure 33). Being memory bound the CPU has to wait for data being written into the cache most of the time. In comparison, wall time is high and decreasing it by increasing the frequency has almost no effect anymore.

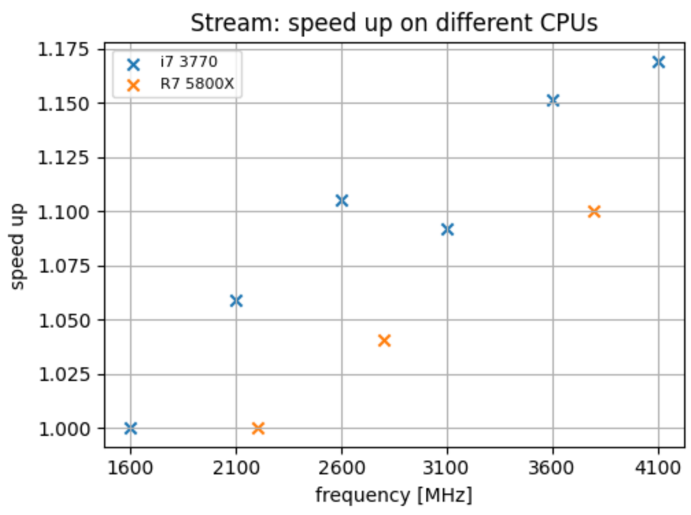


Figure 32: Wall time of i7 3770 and R7 5800X with array size of 51.2 MiB

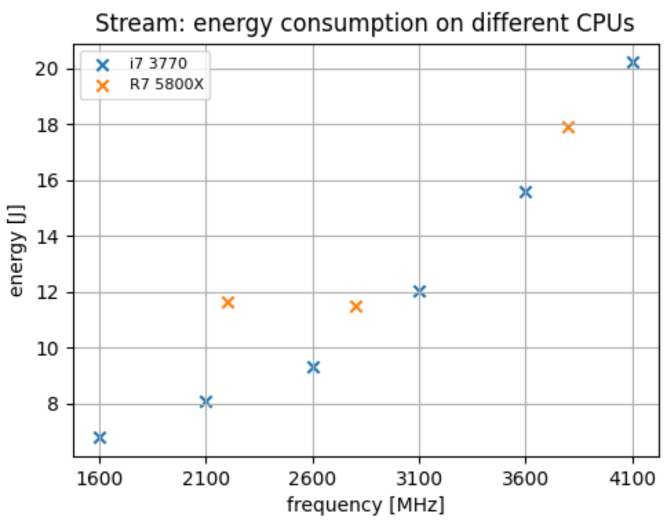


Figure 33: Energy consumption of i7 3770 and R7 5800X with array size of 51.2 MiB

Taking a closer look at the edge cases of the CPUs with an array size of 3.2 MiB for the i7 and 12.8 MiB for the R7 respectively, where it starts to be memory bound, when assigning more threads would provide more L1 and L2 cache and could lead to a higher bandwidth, there are just measurable changes in bandwidth. This supports the assumption that the CPUs are dependent on L3 cache for this benchmark. For this benchmark energy consumption rises with higher frequencies. However, this is not the case with higher thread counts whatsoever (figure 35).

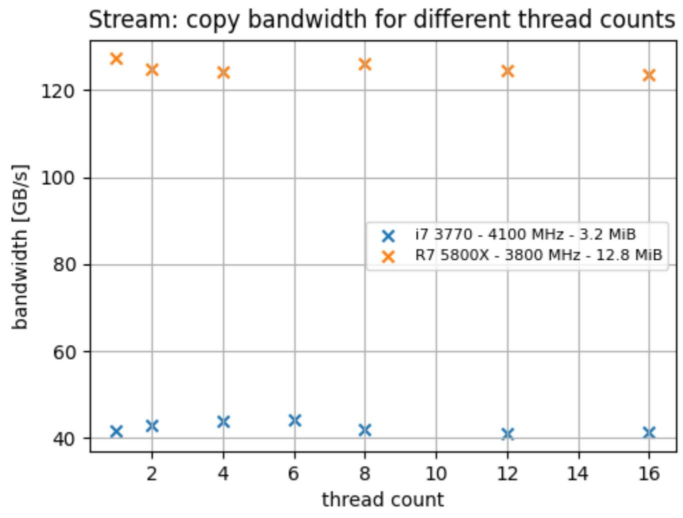


Figure 34: Bandwidths at edge case array sizes with different thread counts

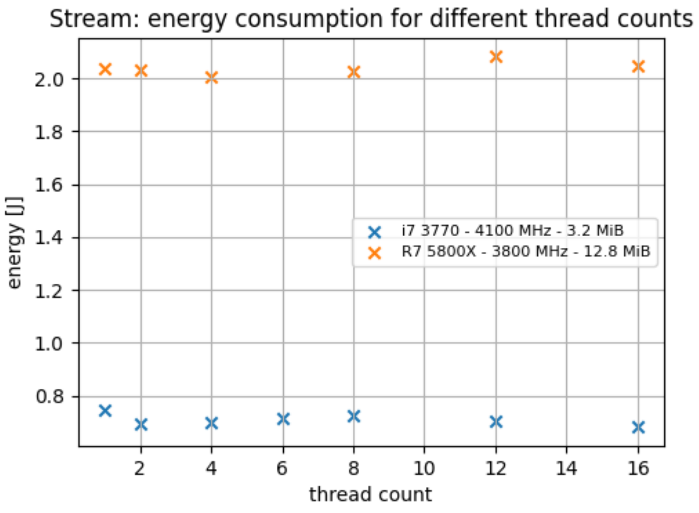


Figure 35: Energy consumption at edge case array sizes with different thread counts

4.4 Vector Multiply-Add

The results for vectorization are retrieved with variable vector sizes and a thread count of 4. Vector sizes were chosen to be in range of the sizes of the caches of the tested systems. The benchmark gets compiled with `-march=native`, `-fopenmp` and `-O2`. When running the benchmark on different vector sizes there is a sweet spot for the R7 5800X regarding speed up at 512 kiB (figure 36). The decline of speed up with larger vector sizes can be attributed to the R7's L2 cache size of 512 kiB per core. On the tested vector sizes the R9 7900X also yields best results with a vector size of 512 kiB. While SSE can achieve a speed up of more than 6 on both systems at this vector size, AVX can achieve even higher speed ups but not double the speed up of SSE, which could be expected, since AVX provides two times larger registers as SSE. For AVX512 it is similar, whose registers have double the size of the registers of AVX, but AVX512 can not yield a corresponding speed up here as well (figure 37). It is worth noting, that in the 128 bit wide XMM registers of SSE two double precision floating point elements can be loaded, which results in a speed up of over 6.

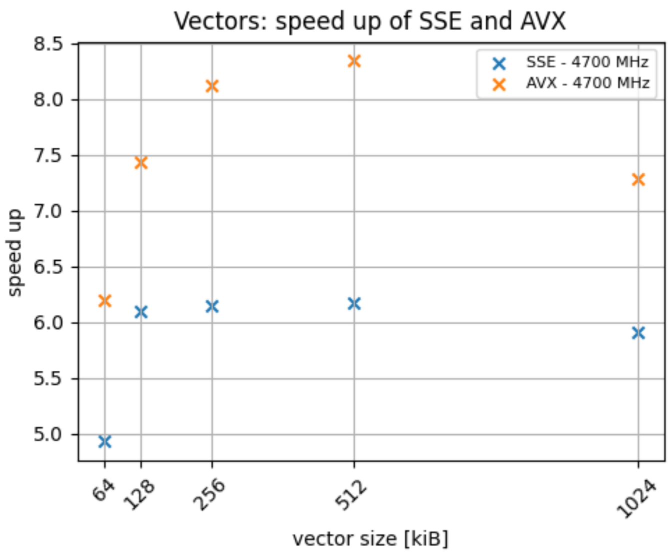


Figure 36: Speed up of R7 5800X using different SIMD instructions sets

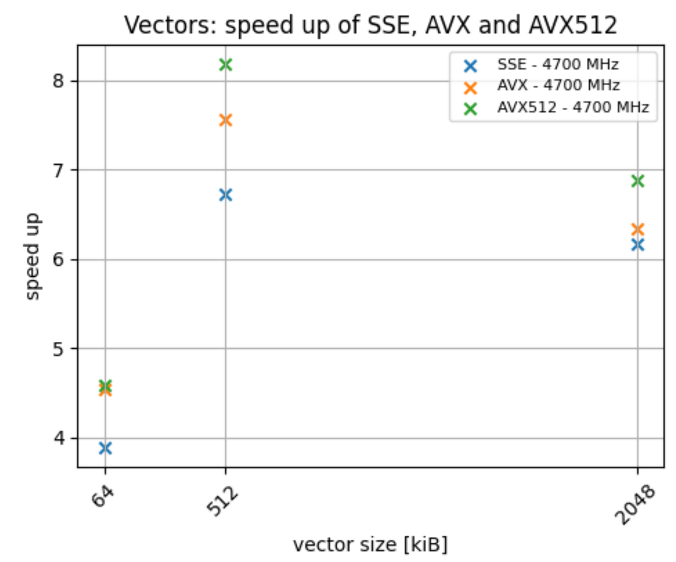


Figure 37: Speed up of R9 7900X using different SIMD instruction sets

In figure 38 speed up is about the same for the i7 3770 with vector sizes of 128 kiB and 256 kiB. Larger vector sizes result in lower speed up, which again can be attributed to the size of the L2 cache, in this case 256 kiB per core. Here the speed up achieved with SSE is only about 5. AVX on the other hand only achieves a speed up of about 4.4 on this system.

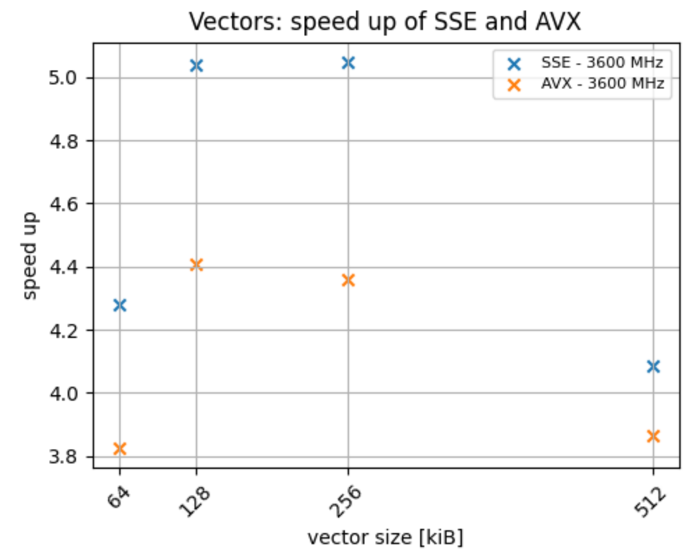


Figure 38: Speed up of i7 3770 using different SIMD instruction sets

The power draw tends do be higher when exploiting vectorization (figure 39), especially when using larger vectors. This can be attributed to the SIMD execution units utilized. For the R7 when using larger vectors an increase of power draw of about 15% can be measured when using vectorization, with smaller vectors the difference diminishes. For AVX with a vector size of 64 kiB the power draw is even lower than when not using any form of vectorization, while with SSE it is higher by about 10 W. With a vector size of 512 kiB the power draw is higher with AVX than with SSE, and with a vector size of 2048 kiB the power draw is roughly the same for both instruction sets.

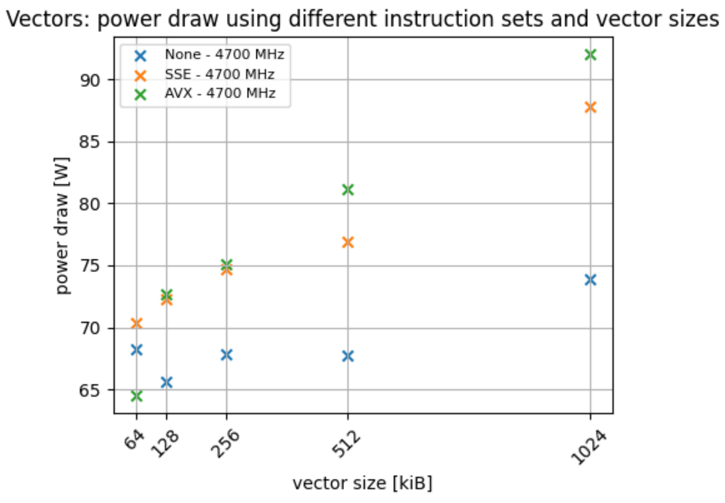


Figure 39: Power draw of R7 5800X using different SIMD instruction sets

For the R9 the power draw increases by 25 to 33% when using vectorization (figure 40). Again the difference gets smaller with smaller vector sizes. For a vector size of 8,192 elements the power draw is lower than when using vectorization, regardless of the instruction set. AVX512 draws the least power for smaller vectors (64 kiB) and larger vectors (2048 kiB) between the different instruction sets. SSE and AVX512 draw more power, while AVX draws less power than not using any vectorization at a vector size of 512 kiB.

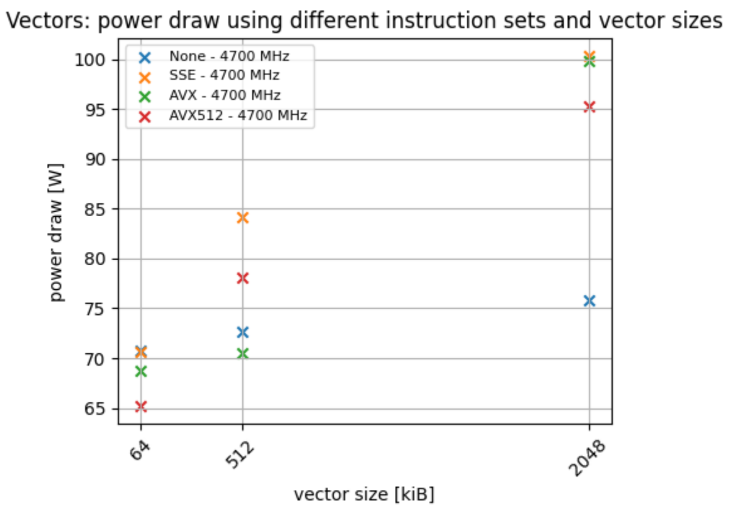


Figure 40: Power draw of R9 7900X using different SIMD instructions sets

The Intel system shows a higher power draw as well when using vectorization. The difference is about 15%, similar to the R7's power draw increase. The power draws of SSE and AVX are roughly the same. With increasing vector size the power draw when using vectorization increases sublinear, while power draw increases superlinear when using no vectorization.

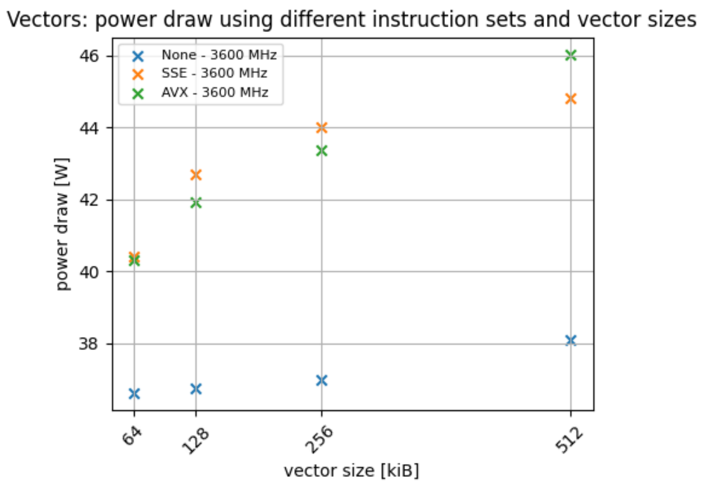


Figure 41: Power draw of i7 3770 using different SIMD instruction sets

The power draw of the R7 when using vectorization may be higher, but the speed up over using none is so significant, that the energy consumption decreases by a factor of about 5 for vector sizes of 512 kiB and 2048 kiB. Even if not identifiable, for a vector size of 64 kiB the power draw is reduced by a factor of about 10. Energy consumption increases linearly with vector size.

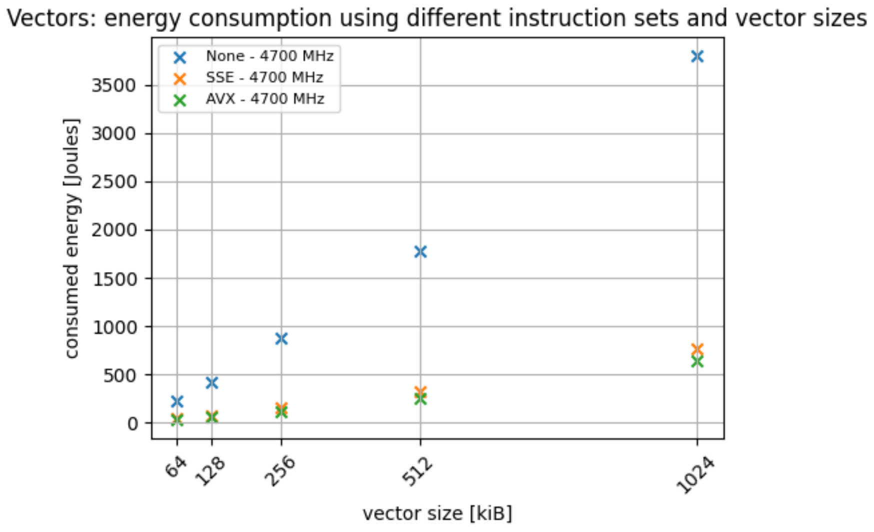


Figure 42: Energy consumption of R7 5800X using different SIMD instruction sets

For the R9 the energy consumption is reduced by a factor of roughly 5 when using vectorization. There is no dependency on the vector sizes the benchmark was executed with.

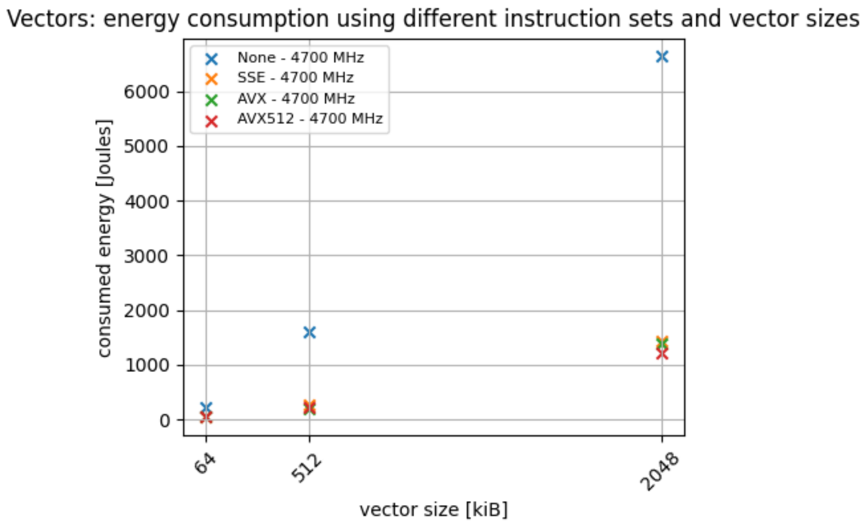


Figure 43: Energy consumption of R9 7900X using different SIMD instruction sets

Energy consumption is 3 to 3.5 times higher when not using vectorization on the i7, again regardless of the given vector sizes.

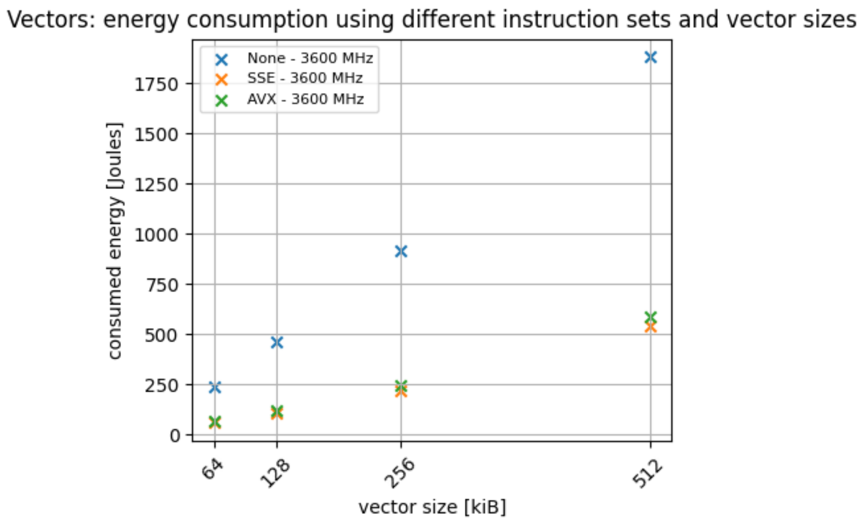


Figure 44: Power draw of i7 3770 using different SIMD instruction sets

When using AVX and AVX512 the results do not scale directly with their respective register size. This could be due to incorrect usage of compiler flags, intrinsics or memory alignment, such that these instruction sets can be ideally exploited.

5 Conclusion

The results of this thesis show that CPUs did not only get faster, they are also more energy efficient for most workloads, despite the recent trend of increased power draw. While it is significantly more energy efficient to utilize as many cores as available or possible, it is not efficient to apply high frequencies. Using SMT can also be advantageous for speed up, and even if it is not, there is no drastic increase in wall times or energy consumption. Memory bandwidth shows a considerable uplift on newer CPUs as well as larger cache sizes reducing the hazard of cache misses in order to stay computationally bound. The use of vectorization shows major reductions of wall times reducing energy consumption as well. While AVX and AVX512 provide larger registers, there was no benefit measured regarding speed up, which would have been expected.

Newer technologies like SMT, AVX or AVX512 are capable of potentially reducing wall times to a fraction. However, software has to be able to exploit these technologies. Software developers have to consider the features CPUs provide and know how to exploit them. Cache sizes also have to be kept in mind, since memory boundness makes the computational performance of CPUs no longer crucial and energy consumption is still increased, when increasing the computational power with higher frequencies.

References

[1] *[PATCH v7 0/4] perf/x86: add Intel RAPL PMU support.* URL: https://lore.kernel.org/lkml/20131112181033.GD3499@tassilo.jf.intel.com/T/. Accessed: 2023-03-14.

[2] BeeOnRope. URL: https://stackoverflow.com/questions/56852812/simd-instructions-lowering-cpu-frequency/56861355. Accessed: 2023-03-14.

[3] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz. "Chapter 2 - High-performance embedded computing". In: *Embedded Computing for High Performance.* Ed. by J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz. Boston: Morgan Kaufmann, 2017, pp. 17–56. ISBN: 978-0-12-804189-5. DOI: https://doi.org/10.1016/B978-0-12-804189-5.00002-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128041895000028.

[4] *CPU frequency scaling.* URL: https://wiki.archlinux.org/title/CPU_frequency_scaling. Accessed: 2023-03-14.

[5] T. Downs. URL: https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html. Accessed: 2023-03-14.

[6] *GCC online documentation.* URL: https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/x86-Options.html. Accessed: 2023-03-14.

[7] *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B.* URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html. Accessed: 2023-03-14.

[8] R. T. Jeff Hammond Paige Weber. *STREAM.* URL: https://github.com/jeffhammond/STREAM. Accessed: 2023-03-14.

[9] K. N. Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: vol. 3. 2. New York, NY, USA: Association for Computing Machinery, Mar. 2018. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[10] T. Klotz. *Bachelorthesis Repository.* URL: https://github.com/TomMahawk711/bachelorthesis. Accessed: 2023-03-14.

[11] M. Larabel. URL: https://www.phoronix.com/review/amd-zen4-avx512/6. Accessed: 2023-03-14.

[12] D. Miralles et al. “Accelerating software radios by means of SIMD Instructions. A case for the AVX2 and AVX512 Extensions”. In: Sept. 2018.

[13] *Power Capping Framework*. URL: `https://www.kernel.org/doc/html/latest/power/powercap/powercap.html`. Accessed: 2023-03-14.