Professur für Höchstleistungsrechnen
Department Informatik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

MASTER THESIS

Automatic Loop Kernel Analysis and Performance Modeling

Julian Hammer

Erlangen, July 30, 2015

Examiner:	Prof. Dr. Gerhard Wellein
Advisor:	Dr. habil. Georg Hager

Eidesstattliche Erklärung / Statutory Declaration

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertig habte und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

I hereby declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Erlangen, July 30, 2015

Julian Hammer

Zusammenfassung

Die Rechenleistung von Programmschleifen ist für numerische Lösungsverfahren auf modernen Computerarchitekturen von besonderer Wichtigkeit. Eine performante serielle Implementierung ist die Grundlage für die effiziente Nutzung moderner paralleler Architekturen. Um dies zu erreichen müssen Anwendungsentwickler größten Wert auf die Modellierung und Analyse der Rechenleistung legen. In dieser Arbeit wird der quelloffene *Kerncraft* Werkzeugkasten vorgestellt, welcher Anwendungsentwickler durch automatische Leistungsanalyse und -modellierung mittels statischer Quelltextauswertung und Leistungsprognosen auf Basis der Execution-Cache-Memory und Roofline Leistungsmodelle unterstützt. Es wird gezeigt, dass die Vorhersagen bei einer Vielzahl von schleifenbasierten Codes, auf verschiedenen modernen Architekturen und in allen Speicherebenen den Messwerten gut entsprechen.

Abstract

The performance of loop kernel codes on modern hardware architectures is becoming ever more important, especially in light of the growing demand for simulations by researchers and engineers. This requires detailed modeling and analysis by the application developer in order to achieve a good single-core performance, which is vital for making efficient use of modern parallel architectures. In this work we present the open source *kerncraft* toolkit to support application developers by automatically analyzing and modeling the performance of loop kernel codes using static code analysis and predictions based on the Execution-Cache-Memory and Roofline performance models. We can show that our predictions match the measured "real-world" results on a wide variety of loop kernel code (stencil and pure streaming codes) on different modern processor architectures, both in-cache and in memory.

Contents

1 INTRODUCTION

Over the last decades computers have become more and more powerful, but on this path computer architects had to overcome many performance barriers with more complex architectures. Some of these enhancements are mostly transparent to the application developers (e.g., caches, branch prediction), but others require understanding and active use by the software developer in order to make use of the advantages (e.g., vector instructions, multiprocessing). This increases the challenge of writing efficient software, because a deep understanding of the hardware architecture is required. With the growing demand of high performance numerical simulations for technical and non-technical applications (e.g., weather simulations, big data, games, video encoding), it is vital to support application developers with tools and means to make efficient use of the underlying hardware architecture. The "hot spots", where most of the runtime is spent, need to be tuned to reduce cost in terms of machine occupation and energy consumption. This tuning requires an in-depth understanding of the targeted architecture and is often a non-trivial task. An optimizing compiler is a valuable asset to help the programmer obtain more efficient code, but it is still necessary to know whether there is room for improvement.

Most algorithms in scientific computing are implemented as sequences of loops, which are frequently called *loop kernels*. Often such loops are nested within other loops, then the term loop kernel sometimes refers to the innermost loop body alone. In the simplest case, a loop kernel reads data sequentially from an array and writes updated values, with no temporal locality. A simple triad streaming example is found in Listing 1.1, where array `a` is updated by multiplying `c` with `d` and adding to array `b`. We will call such kernels *pure streaming kernels*.

```
for(i=0; i<N; ++i)
    a[i] = b[i] + c[i] * d[i]
```

Listing 1.1: Basic Triad Streaming Kernel

Stencil codes are another relevant variant of loop kernels. They emerge from, for instance finite-differences discretizations of differential operators when solving partial differential equations. A stencil applies an update pattern to all elements of an n-dimensional array by referencing neighboring elements, with temporal locality. Such an update pattern is called *stencil* and describes how the center element is updated based on its neighboring elements. A typical example, arising from the heat-equation, is presented in Listing 1.2.

The exemplified stencil code is often referred to as “2D-5pt stencil”, since it operates on a two dimensional data set and each update touches five neighboring points (one center for write and four direct neighbors for read).

```
for(j=0; j<N; ++j)
    for(i=0; i<N; ++i)
        a[j][i] = 0.25 * (            b[j-1][i] +
                          b[j][i-1] +             b[j][i+1] +
                                      b[j+1][i]                 );
```

Listing 1.2: Basic 2D-5pt Stencil Code

Neighbors do not necessarily have to be direct neighbors and in that case, such stencils are referred to as long-range stencils and require more attention with respect to cache usage. Stencil codes are of high interest in scientific computing because of their wide applicability in numerical solvers. Loop kernel codes are more generic and allow for patterns which are not restricted to neighborhood relationships, e.g., such as the ones found in vector summation.

A visual representation of the mentioned examples is given in Figure 1.1. We can see the geometric relation of accessed cells to the iteration center in the stencil case. This is the defining information of stencil codes and streaming kernels usually access the data without any offsets.

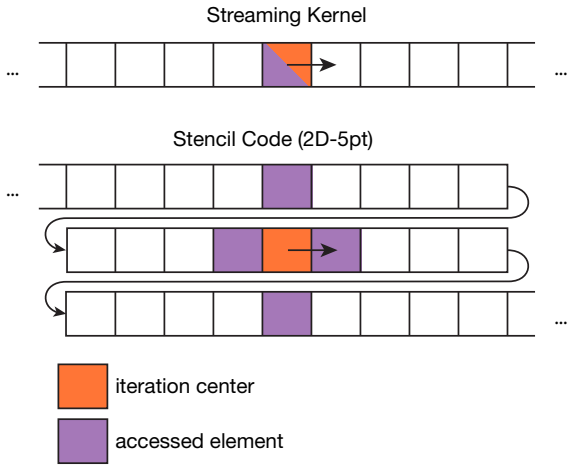


Figure 1.1.: Visual representation of an array from stream kernel in Listing 1.1 and of array `b[][]` from stencil code in Listing 1.2.

Many approaches have been developed to aid performance optimizations by application developers, which are often domain specialists without a fundamental background in computer science. These aids can be either theoretical or practical in nature. Theoretical refers to taking a simplified model of the architecture and comparing it in a pen-and-paper way to the algorithm, e.g., the Roofline Performance Model [1]. The results of a theoretical analysis can then be used as guidance to manually modify the algorithm towards a more efficient version, but it does not yield enhancements by itself and a fundamental understanding of the analysis is required. A practical approach relies on compilers, libraries and tools to make decisions for the developer. This requires that the code can be understood by a black-box tool and can automatically be transformed into a more efficient representation of the original algorithm.

The Execution-Cache-Memory (ECM) performance model by Treibig and Hager [2], Hager et al. [3] and Stengel et al. [4] is another theoretical aid that has recently been developed. It is a refinement of the Roofline model [1], but it uses more technical information about the underlying hardware to arrive at a runtime prediction, rather than relying the prognosis on benchmarks. The resulting prediction is not a single number, but shows the individual contributions to the runtime of a loop kernel from memory hierarchy levels, as well as arithmetic operations. A detailed description of the Roofline and ECM models will be given in Chapter 2.

1.1. Motivation

1.1.1. Performance Modeling

Performance modeling is the process of predicting the performance or runtime of code on a hardware platform. Since modern hardware architectures are very complex, in theory, a perfect prediction would require the code to be executed on the same hardware and with the same input. This would eventually not be a simulation or modeling approach anymore, but a benchmark measurement. To reduce the complexity, simplifications and abstractions need to be made in such a way that the result of the model will yield a close to real prediction, without making the model inapplicable because of its complexity. The choice of simplifications and abstractions lies at the core of any performance model.

In this work we will restrict ourself to analytical modeling of steady-state loop kernels using a white-box approach in regard to the underlying kernel code and in particular to two performance models: ECM and Roofline. These models bring already all the simplification and abstractions with them, but application to a specific code and hardware requires additional work. From here on, we will refer to the application of a performance model onto a software code and hardware architecture as *performance modeling*.

Performance modeling is essential for different use cases: to predict the runtime on a larger system, to asses the expected speedup on a new hardware architecture, to estimate the required hardware to find a solution within a fixed timeframe, to know if the code is at its optimal performance and many more. Benchmarks can give some insight, but are non analytical and can not be applied to theoretical changes in hard- and software. Another problem with benchmarks is the cost of machine usage on a large scale, which may hinder the developers to run enough to get a good insight.

1.1.2. Automation

Performance modeling can be very tedious, since one has to identify the code section where most of the runtime is spent and then analyze that section thoroughly, in order to understand exactly the number and type of arithmetic operations as well as memory access patterns. Nonetheless, this process rewards with the deep understanding required to optimize the performance. It has been shown that the compiler—as the most common practice tool—fails to achieve the best possible performance in many and even simple cases as shown by Gropp [5], thus performance modeling needs to be simplified and made more accessible without compromising its usefulness by hiding the analysis process. After gathering all the performance relevant information, it needs to be analyzed with respect to the targeted hardware architecture. This requires information about the central processing unit (CPU) and underlying memory hierarchy with their theoretical and benchmarked performance data. In case of non-stencil and non-streaming codes this analysis can be arbitrarily complex, which is why we apply certain restrictions on the given code.

For someone without experience or background in computer architecture, this can be a tremendous task. A tool was required to take over the tedious parts and make performance modeling more accessible to a wider audience. The goal is to automatically analyze a hot section of code, typically a stencil kernel, and extract the memory access patterns as well as arithmetic operations. This information is then used as input for the performance model to eventually yield a runtime (or performance) prediction. The hardware information is a separate input and can be collected semiautomatically. A small collection of common architectures and kernel codes is part of this work and will be available to potential users.

Automation will not only make performance modeling faster, it will also reduce the amount of knowledge (hardware specs) that users needed to be able to apply a model to a code. These simplifications do not mean that less insight is necessary. Because, in order to interpret and make sense of the resulting output, a good grasp of performance modeling and hardware architectures in general will always be required.

1.2. Task

Our objective is to create a tool which automatically analyzes a stencil code loop nest or streaming kernel, retrieves all performance relevant information about it and applies the ECM and Roofline model for a given hardware architecture. Only single core predictions are aimed for, but core saturation is also of interest. Symmetric multithreading (SMT) and scaling beyond one chip will not to be investigated in this work.

Loop kernels are given in C-code, but certain restrictions are imposed:

- all access to arrays is done using multidimensional notation (e.g., `a[i][j]`)
- the code may only use for-loops with constant initial value and known iteration length and step size (e.g., `for(int i=1; i<23; ++i)`)
- all arithmetic operations need to take place in the innermost loop
- no other statements (e.g., function calls) are allowed

A more detailed set of applying restrictions is given in Chapter 3. The tool has to work with a wide variety of stencil and streaming codes, of which a representative collection will be assembled and used for testing purposes in this work. These codes, as well as the tool itself will be available for download[1].

The targeted hardware platforms are architectures with inclusive and write-back caches and need to be supported by the Intel Automatic Code Analyzer (IACA). The hardware description is compiled from benchmarks and data sheets. Some of it can be gathered automatically and a tool to do so is provided, as well as a set of machine description files for future reference.

Finally, the implementation of the ECM model needs to be validated on several hardware architectures and the performance predictions compared to actual measurements. The toolkit needs to be released under an open source license to allow other researchers to make use of it.

1.3. Related Work

In their paper “Reuse Distance Analysis”, Ding and Zhong [6] developed an efficient method for gathering cache access patterns from live running codes using statistical sampling. This allowed them to have arbitrary code run at almost native speed, while collecting this vital performance information. Similar work was done in “Pinpointing data locality bottlenecks with low overhead” by Liu and Mellor-Crummey [7]. In contrast, the tool developed in this work never actually executes the code under investigation (except for validating predictions), which has the natural performance advantage and the additional benefit of providing more insight into the origin and relation between data accesses. On the other hand, our analysis can only be applied to code fulfilling the restrictions mentioned in Section 1.2.

Lo et al. [8] published in “Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis” an approach which automatically collects benchmark results by sweeping over the whole memory hierarchy with different working set sizes. In this respect, their approach resembles the method used in this work for collecting machine-specific input data. As before, the results presented are based on benchmark runs, but here, the authors go further and also look at GPUs and many core processors.

In “Generating Performance Bounds from Source Code” Narayanan et al. [9] analyze simple kernel codes directly from the source code and similar information is extracted by executing the code without any computation while intercepting data accesses. However, they are missing the above mentioned reuse distance or data access pattern, which are vital to predict latency variations arising from cache vs main memory access.

[1] `https://github.com/cod3monk/kerncraft`

1.4. Results

We present the Kerncraft toolkit, which combines a parameter file describing the hardware architecture and information gathered from the kernel code to a suitable performance prediction. We can show that the predictions through the ECM model are very accurate for most test-cases on the latest three microarchitectures by Intel: Sandy Bridge, Ivy Bridge and Haswell. The twelve test-cases include four typical streaming benchmarks (DAXPY, add, scale and triad), two reduction algorithms (scalar product, vector sum) and six stencil codes. The prediction of main memory access is more difficult with the later two microarchitectures and correction factors had to be employed to produce acceptable results. All predictions have been validated by benchmarks, which were generated from the same kernel code as used for predicting the behavior.

1.5. Outline

The underlying ECM model and an example analysis of the computer architecture will be presented in Chapter 2. In the subsequent Chapter 3, the general design and algorithmic details of the developed tool are explained. The evaluation in Chapter 4 compares predictions based on the ECM model with obtained measurements on three different hardware architectures. In the final Chapter 5, possible enhancements and their usefulness to application developers are discussed.

1.6. Acknowledgments

I would like thank my advisors Georg and Jan for their support and help, especially Georg for his meticulous attention to detail and helpful comments to the last minute, and Jan for great practical suggestions and for providing the LIKWID toolkit. I am also grateful to Daniela and Balthasar for helping me iron out my awkward formulations and typographical mistakes, and also suggestions on my visualizations, so that this work is more pleasant to read. The Fachfachtsinitiative Computational Engineering supported me by giving me a space to work in and always something[2] to distract myself with, if it was necessary to clear my head.

[2] everything from geeky discussions to cold beer

2 BACKGROUND

In this Chapter we give a general overview of the theory and technology on which this work is based and go into detail about the specific computer architectures, performance models, applications and libraries used.

2.1. Computer Architecture

Modern mainstream computer architectures are still exclusively based on the von Neumann architecture [10]. Many additions have been made, but the main components that we need to consider are: the *arithmetic logic unit* (ALU) responsible for calculations, the *control unit* (CU) taking care of control flow of the program and the *memory* supplying the CU and ALU with input data and storing output data. Figure 2.1 provides an overview of the common components. In this Section we will take a look at the execution side, in von Neumann's terms: ALU and CU, and the memory side in modern Intel x86 multicore processors. We will specifically target the Sandy Bridge microarchitecture and mention differences to newer Intel microarchitectures relevant to this work in Section 2.2.

We will beginn with an an overview of single-core and single-thread features of modern computer architectures. Further on we will take a look at multicore features.

2.1.1. Execution

The time for code execution (i.e., not counting the data transfers from the memory hierarchy) is governed by the processor's clock speed: the higher the frequency the faster the execution. Consequently, the CPU vendors tried to increases the clock speed with every new model over several decades, but due to energy and thermal constraints this performance screw became less attractive and the increase subsided in the mid 2000s. In many cases newer processors even decreased the clock speed compared to older models in the last decade, to increase energy efficiency. Other means had to be found to guarantee a steadily

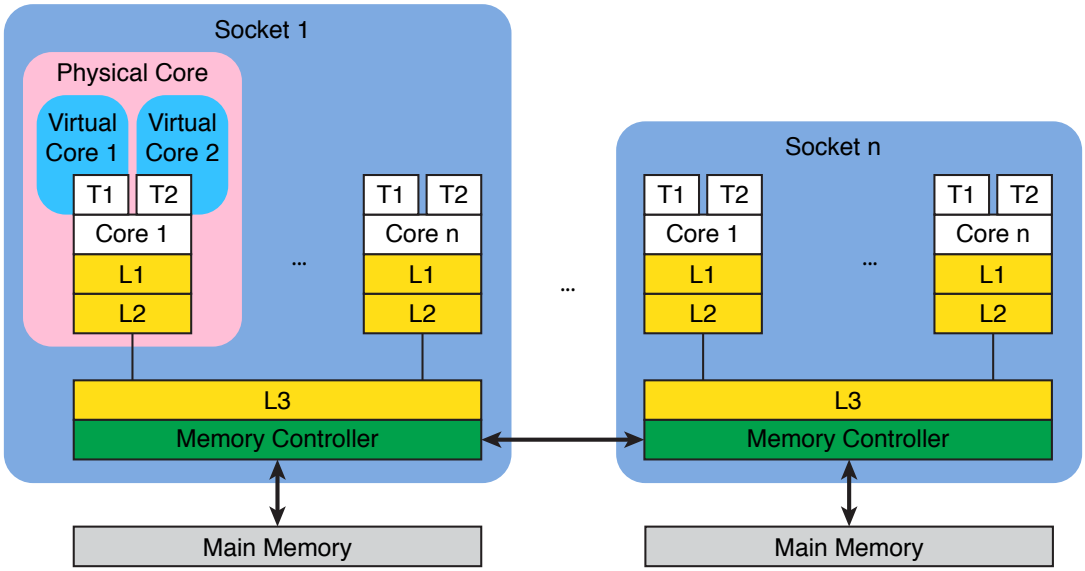


Figure 2.1.: Overview of a modern computer architecture with two threads per core, three cache levels (two private and one shared) and an interconnect between chips (or sockets).

rising application performance across CPU generations. We will have a look at the specific architectural enhancements that improve performance at constant clock speed—for numerical applications—present in current processor architectures.

Micro Operations

x86 is a *Complex Instruction Set Computing* (CISC) architecture, which means that instructions and operands can vary in encoding length and execution time. Internally it is implemented in a *Reduced Instruction Set Computing* (RISC) way, which is why *micro operations* (µops) were introduced. CISC operations (macro-ops) need to be decoded and split into RISC µops on the fly, in hardware. The translation to µops allows the efficient implementation of a pipeline concept without breaking support for the x86 instruction set, which was meant to simplify the work of low-level assembly programmers and reduce code size.

Decoding is done depending on the instruction and most instructions can be translated to single µops, but more complex instructions need to be translated to up to four µops [11]. This is done in multiple decoders in parallel which can queue several µops per cycle, see Figure 2.2.

Pipeline

Pipelining is done by splitting operations that need more than one cycle to execute into separate stages. The slowest stage will dictate the overall speed at which instructions can proceed within the pipeline. Each stage is used by a different instruction, and after an instruction is processed by all stages, its result becomes available. The advantage is that more instructions can be handled in parallel, but only if the pipeline can be filled with suitable instructions.

Due to the pipeline concept in modern CPUs—in particular in all current Intel CPUs—there are two different types of cost associated with all instructions: latency and throughput. *Latency* is the number of cycles it takes to execute a single instruction until its result becomes available for further processing. This is the length of the pipeline. *Instruction throughput* is the number of cycles before the same instruction can be

issued again[1] and it is the time it takes an instruction to move from one stage to the next in the pipeline. Henceforth, we will refer to instruction throughput as *throughput*. Throughput can only be used as a valid metric if a sufficient number of operations can be dispatched. This is usually the case with loops that have many iterations, such as stencil codes that update a large number of grid points.

Out-of-order Execution

To efficiently fill the pipeline, the core supports out-of-order scheduling of µops through reordering and utilizing several execution ports. Execution ports are responsible for certain functionality and their number and function depends on the specific microarchitecture and varies across processor generations. For instance, the Sandy Bridge architecture has six ports: 0, 1 and 5 are responsible for arithmetic operations, 2 and 3 for data loads and 4 takes care of data stores. An overview of the functional units behind the execution ports on Sandy Bridge can be found in Figure 2.2. At each cycle the scheduler may dispatch one µop to each available port, so the µop throughput limit in this part of the architecture is six µops per cycle.

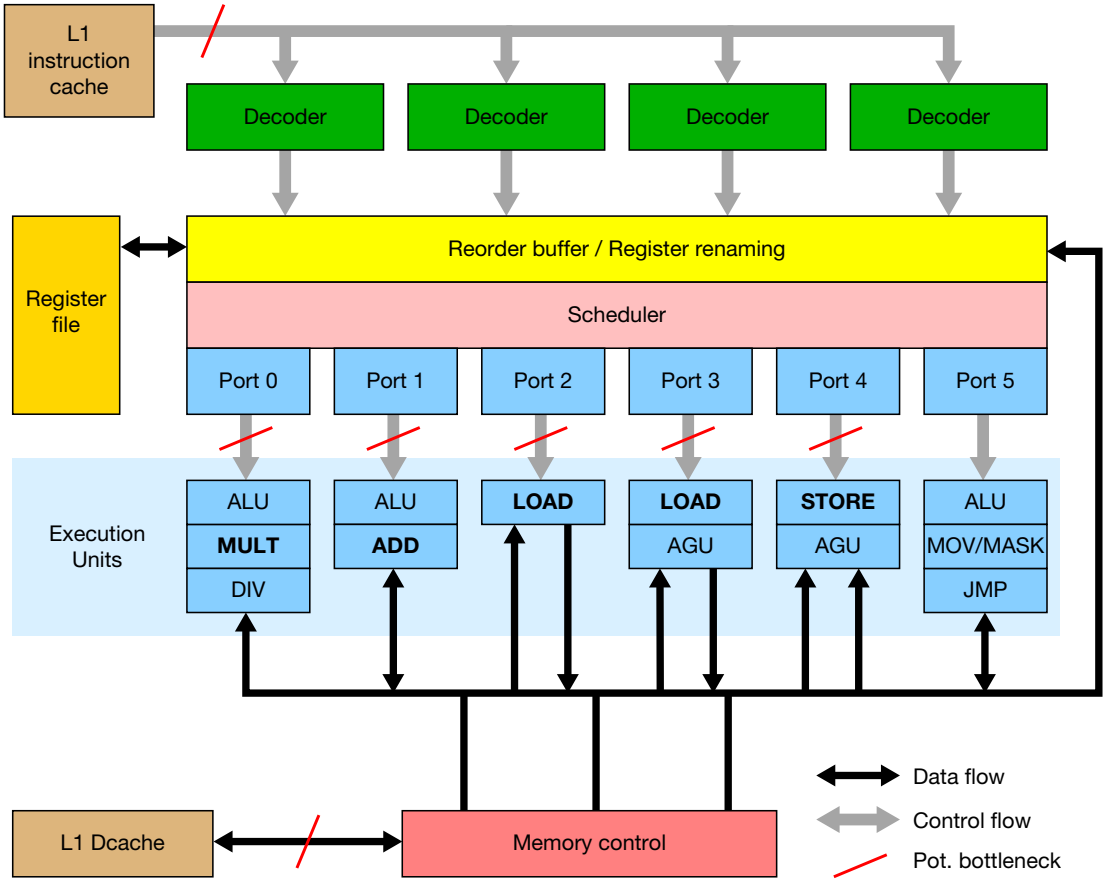


Figure 2.2.: Execution Pipeline and Ports of a single Intel Sandy Bridge Microarchitecture core (based on illustration by Georg Hager)

[1] The definition of throughput and latency is taken from the "Intel 64 and IA-32 Architectures Optimization Reference Manual" [11]. Throughput can also be defined as operations per cycle, which is inverse to our definition and can therefore easily be distinguished.

Reordering of µops is made possible by the renamer, reorder and scheduler components (see Figure 2.2). The renamer prepares µops by renaming data sources and destinations, allocating resources like buffers and assigning them to an execution port. It can understand and utilize the independence of subsequent instructions referencing the same memory locations (i.e., if data is overwritten, the original content may be ignored). The scheduler queues µops and dispatches them to the assigned execution port once all source dependencies have been fulfilled and required resources are available. Retirement takes care of the resulting data and possible exceptions. All instructions are retied in program order, as not to alter the programs behavior by this process.

Arithmetic Operations

On Sandy Bridge and Ivy Bridge CPUs, double precision floating point addition (multiplication) operations can be dispatched to ports 0 and 1 with a latency of 3 cycles (5 cycles) and a throughput of 1 cycle. This means that in every cycle a floating point multiplication can be executed, but it will take 5 cycles before the result becomes available. Divisions, on the other hand, have a latency of 27-35 cycles on Ivy Bridge and 33-45 cycles on Sandy Bridge and throughput of 28 cycles on Ivy Bridge and 44 cycles on Sandy Bridge, and so almost no pipelining is done here. Square root instructions latencies are also high due to their iterative nature of computation: a latency of 21-45 cycles on Sandy Bridge and 19-35 cycles on Ivy Bridge occurs and it takes of 20-42 cycles on Sandy Bridge and 28 cycles on Ivy Bridge before another instruction can be issued. This means that division and square root operations should usually be avoided, even if it means to introduce many more multiplication instructions into the algorithm, unless the devision can be executed in parallel without delaying other computations.

Single-Instruction-Multiple-Data (SIMD)

Another way to improve parallelism and performance is to use single-instruction-multiple-data (SIMD) [12] features. SIMD instructions allow the execution of an single operation on multiple elements in a special purpose vector storage. Different implementations pursuing this principle have been around for decades, but here we will focus on the Advanced Vector Extension (AVX), an instruction set first introduced by Intel in the Sandy Bridge architecture in 2011, extending the Streaming SIMD Extension (SSE), which was already available since 1999.

The usage of this vectorized floating point arithmetic instruction set is essential for numerical codes to achieve good performance on modern processors. It allows the processing of 256-bit long data stream containing four double precision or eight single precision floating point numbers with single arithmetic instructions.

Loops and Jumps

We have learned that the pipeline concept requires a continuous stream of instructions to perform well, but what happens if a conditional branch (or jump) is part of the instruction stream? Loop constructs are ubiquitous in numerical codes and result in an conditional branch at its end. Performance-aware programmers will try to avoid all braches in the innermost-loop, to allow efficient use of CPU pipelines, but the end-of-loop conditional branch is impossible to circumvent. Processor vendors have approached this problem by including *branch prediction* logic in the hardware.

Branch predictors have been an essential component of processors for some time, but are still being refined and tweaked, for a simple reason: they keep the pipeline filled and if the branch target is predicted incorrectly, work and power is wasted and already computed outputs need to be ignored. However a misprediction is usually as bad as no prediction at all, thus even always predicting that a branch is going to be taken can yields quite good results.

In typical loop based codes it can be assumed that the end-of-loop branch is taken most of the time, since iteration lengths are expected to be large. Thus a positive prediction rate of almost 100% can be expected,

unless other branches are taken within the loop (e.g., branches taken based on the calculated data). For this reason, the end-of-loop branch can be ignored in loop-kernel performance analysis.

Multi-Core and Multi-Threading

To increase the performance and power-efficiency of modern processors vendors have introduced multi-core CPUs. This means that multiple computing cores are combined on one chip. Some resources are shared between the cores, e.g., the memory and PCI-express interfaces, but other resources are dedicated, such as executions units and pipelines. A chips performance and power consumption increases linearly with the number of cores, as long as the frequency is constant. This is a vast improvement in efficiency over a single-core CPU where the frequency is increased to achieve the same performance, since the CPU voltage needs to be increased which leads to a much higher power consumption and thermal output.

Multi-threading on the other hand is a way to increase the utilization of out-of-order execution features. It simulates two or more *threads* (or virtual cores) per physical core. The operating system will then make use of all threads using independent processes, which makes it easier for the hardware to fill all execution unites with independent instructions and hide latencies due to memory accesses.

2.1.2. Memory

Not only arithmetic operations, in the sense of floating point operations (FLOP), are relevant, because the data needs to come from somewhere and the results must eventually be written to somewhere. In this Section we will have a look at the LOAD and STORE instructions and the underlying memory hierarchy from main memory through shared and private caches up to CPU registers, as well as the logic associated with it. A schematic view on the complete memory hierarchy of a Sandy Bridge CPU can be seen in Figure 2.3.

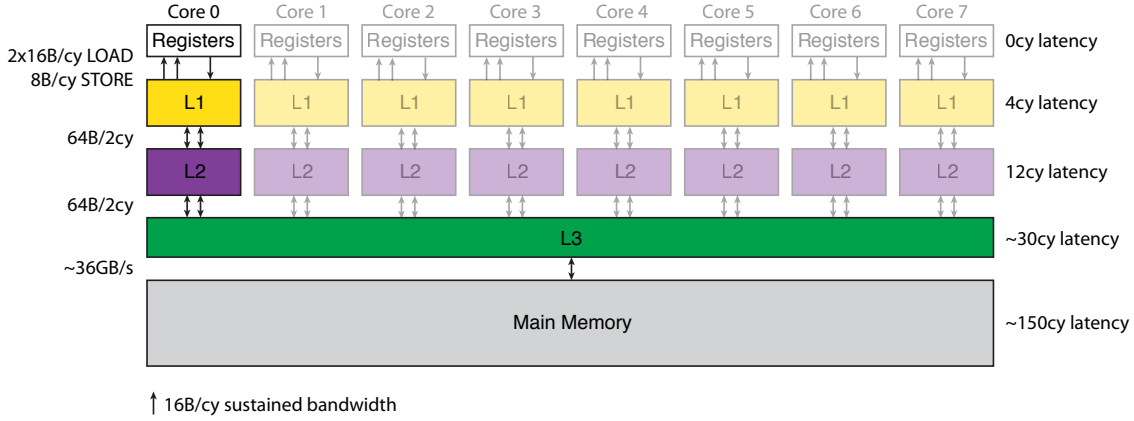


Figure 2.3.: Memory Hierarchy on Single-Socket Intel Xeon Sandy Bridge E5-2680 CPU with 2.7 GHz

CPU Registers

Registers are the storage locations that instructions operate on, both for source and destination data. They are the fastest, smallest and lowest latency storage units available in computers. To get any data into registers from main memory LOAD instructions are used. Vice-versa, STORE instructions write data from registers to main memory.

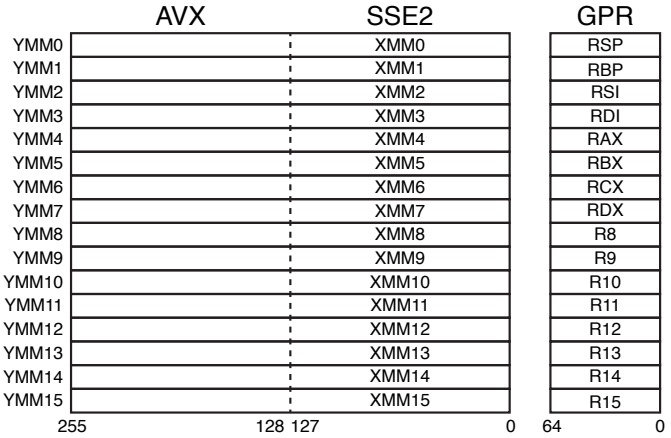


Figure 2.4.: Registers available for floating-point arithmetics on Sandy Bridge CPUs. YMM0-15 are 256-bit vector registers and can only be used in combination with AVX instructions, their lower 128-bit are also addressable via XMM0-15 for use with SSE and SSE2 instructions. The 64-bit general-purpose registers (GPR) are available on all x86 64-bit processors and can be used with all x86 instructions, but the first four (RSP, RBP, RSI and RDI) are usually reserved for special functionality related to memory managment.

Sandy Bridge CPUs have sixteen 64-bit general-purpose registers and sixteen 265-bit vector registers for use with floating-point arithmetic. An overview of all registers related to floating-point arithmetics is shown in Figure 2.4. Due to constraints in number and size, it is usually impossible to load all data into registers and write back the final result after all computations have finished. Thus new data needs to be loaded and results written back to memory all the time. All source data needs to pass through LOAD instructions and destination data through STORE instructions. On Sandy Bridge the load/store throughput is limited by two bottlenecks: number of ports and adresse generation units (AGU). There are two AGUs, thus only two accesses can happen in the same cycle, because only two memory addresses can be calculated, and there are two load ports (2 and 3) and one store port (4) available for instruction scheduling. Thus one 256-bit AVX-load instructions and one 128-bit AVX-store can be scheduled on every cycle or two 256-bit AVX-loads and so on. The overall latency will vary with the location of the data within the memory hierarchy (with main memory having the highest latency and being the furthest away). If data is present in the first level cache (L1), the latency is 5 cycles for LOADs to SIMD and floating-point registers and 4 cycles in all other cases.

Main Memory

The main memory—often just memory—is several orders of magnitude larger than the registers, usually from 8 GB on current notebooks up to 256 GB and more on servers. It is connected to the CPU via a memory controller, located within the CPU packaging on modern processors. On systems with multiple CPUs or sockets, multiple such memory controllers are present, but all memory is logically arranged in a single address space. The data stored in memory associated with other CPUs can be accessed transparently through a CPU-interconnect, although a latency and bandwidth penalty will apply in this case. This concept and resulting effect is called *non-uniform memory access* (NUMA). It is important to restrict accesses to a processor's own memory whenever possible, by applying the "first touch" rule [13].

Caches

Since the main memory needs to be large and affordable, it is not fast enough to supply the CPU registers with data to fully utilize the arithmetic capabilities. Several layers of caches are present to uncouple fast registers from slow main memory, without compromising on the overall available storage capacity. Caches

are several times faster than the main memory and can handle much more data than registers. They are usually located directly on the CPU chip, yielding good latencies. Depending on the cache-level, the size varies from tens of kilobytes to tens of megabytes. An overview of the cache characteristics of recent Intel architecture generations can be found in Table 2.1. In this case it must be noted that the sizes mentioned in the table are not specific to the microarchitecture, but depend on the exact model. In Table 2.1, the bandwidth is given inverse with cycles per cache-line (cy/CL), since caches do not operate on individual bits or bytes, but in a cache-line granularity and we are interested in the number of cycles it takes to process a cache-line. A cache-line has the size of 64 Byte on current Intel processors and is synonymous to memory blocks on which main memory operates.

Caches operate transparently from the programmer's point of view and can only be detected by performance measurements. There are no special instructions to change their behavior (exceptions are explained in the following section). The fundamental functionality of a cache is rather simple: on a LOAD, data is requested from the *first level cache* (L1), but if it is not found there the request is forwarded to the second level cache L2 and so on, until the cache-line which contains the requested data is retrieved from the main memory into the *last level cache*. The result is that each cache-level is then retaining a local copy of the cache-line for faster subsequent accesses. This method, used by Intel, is called inclusive caching, because all data present in L1 is also located in L2 and L3.

Due to the nature of caches, they are always fully occupied. The location where a new cache line is to be placed in cache is decided by the *cache replacement policy*. We need to know that cache-lines can not be placed at arbitrary positions in a cache. They are restricted to certain cells, as this makes locating them fast and efficient. The number of cells available to one memory location is called associativity and depends on the placement-algorithm implemented in the CPU's hardware.

A typical replacement policy is the least-recently-used (LRU) approach, where the cache-line within the set of possible storage location is replaced that has not been accessed for the longest period. Many other policies have been studied and are in use on other architectures, ranging from random-replacement (RR) to first-in-first-out (FIFO) [14]. All of them have their advantages and disadvantages, depending for example on the effort to implement in hardware and the software application that is using the cache. Intel describes their cache-replacement strategy as pseudo LRU (PLRU), but does not go into detail about the exact algorithm used [15].

After data has been modified in the cache it needs to be written back to memory, this is known as *evict*. It usually happens when a cache-line is replaced to make space for other data. Evicts have to pass through all cache-levels from the CPU to the main memory and are done in cache-line granularity, just like when data is loaded into a cache.

Since individual elements in a cache-line can be modified, but only complete cache-lines may be evicted, it is necessary to load the complete cache-line into L1 before any modifications can be made. This procedure is called *write-allocate* and leads to additional traffic between caches and memory. It is possible to circumvent this behavior by utilizing special **non-temporal store** instructions, but this is only useful if complete cache-lines are modified independent of their previous content. Non-temporal stores are usually not used by compilers automatically and thus require the programmer to actively make use of that feature.

Cache-Coherency

If caches contain a subset of the information located in main memory and a cache can be private to cores, how can it be acceptable that other cores cache the same information. In case of LOAD—or read—access this pose a problem, since caches can have simultaneous copies of the same cache-line to speed up local read access, but if the data was modified in another cache and not yet evicted, the memory contains the wrong data. That is when the *cache-coherency* comes into play, making certain that all cores operate on the same coherent data set, by invalidating or evicting other cached copies. Also, if a cache-line is located in multiple caches and it is modified by one of the cores, this information needs to be passed on to all other cores that have a copy of the cache-line, to invalidate or update their copy. This can lead to significant performance degradation if multiple cores access and write to the same cache-line frequently.

	microarchitecture	Sandy Bridge (SNB)	Ivy Bridge (IVB)	Haswell (HSW)
	year of release	2011	2012	2013
	model name	E5-2680	E5-2690 v2	E5-2695 v3
	base clock speed[a]	2.7 GHz	3.0 GHz	2.3 GHz
	cores	8	10	14
	cacheline size	64 B	64 B	64 B
L1 cache	size	32 kB	32 kB	32 kB
	shared	no	no	no
	inverse bandwidth to L2	2 cy/CL	2 cy/CL	2 cy/CL[b]
L2 cache	size	256 kB	256 kB	256 kB
	shared	no	no	no
	inverse bandwidth to L3	2 cy/CL	2 cy/CL	2.8 cy/CL[c]
L3 cache	size	20 MB	25 MB	35 MB
	shared	yes, by all cores	yes, by all cores	yes, by all cores
	inverse bandwidth to mem.	up to 40 GB/s	up to 48 GB/s	up to 54 GB/s
	execution ports	6	6	8

[a] The actual frequency can be higher than the base clock speed with turbo mode, which was disabled for all measurements.
[b] According to official documentation one cache-line can be loaded per cycle, but Hofmann [16] has shown that in practice only 32 B can be loaded per cycle.
[c] This inverse bandwidth is only valid in the single-core case, due to uncore frequency scaling (frequency increases and inverse bandwidth decrease with single-core). With two ore more cores in use, this inverse bandwidth decreases to 2 cy/CL [16].

Table 2.1.: Intel Xeon Models used for Validation

Prefetching

Prefetching is a way to bring data into the cache that is to be accessed in the near future. This leads to lower latencies of LOAD and STORE instructions. Prefetching can be initiated by hardware, transparent to the programmer, or by software, explicitly by the programmer or compiler. Hardware prefetchers are based on algorithms monitoring access patterns and, predicting future access, this can be as simple as always loading adjacent cache-lines or as complicated as tracking accesses and deducing stride lengths from that. The implementation of hardware prefetchers in current micro processors is usually not revealed by vendors. To make use of hardware prefetching, a good approach is to access sequential elements in a stream-wise fashion. If hardware prefetching fails, it is possible to make use of special prefetch instructions.

2.2. Testbed Hardware

In Table 2.1 we have put together an overview of the main metrics describing the three most recent microarchitectures from Intel: Sandy Bridge, Ivy Bridge and Haswell. Some of the information depends on the specific model and not solely on the microarchitecture.

The changes between Sandy and Ivy Bridge are small, since Ivy Bridge was developed in the *Tick* cycle of the *Tick-Tock* development model followed by Intel since 2007. Tick means that a new (smaller) fabrication process is introduced, but CPU features are not considerably enhanced. This is what was done during the transition from Sandy Bridge to Ivy Bridge. During *Tock* cycles, the fabrication process is kept the same, but substantial changes are made in the microarchitecture. The transition from Ivy Bridge to Haswell was such a Tock cycle. As the name of the development model implies, the two cycles always alternate on a yearly basis. Thus Haswell has the most differences in microarchitecture. We will have a look at three differences: uncore frequency, execution ports and AVX2 with fused-multiply-add (FMA) support.

The Haswell microarchitecture offers additional execution ports, allowing more parallelism and out-of-order execution. An additional integer arithmetic unit and address generation unit are associated with

the new ports, others were moved to reduce dependencies. Another improvement—according to Intel's documentation [11]—is the doubled bandwidth between L1 and L2. Unfortunately were we unable to reproduce one cycle per cache-line or 64 Byte per cycle of L1-L2 bandwidth [16]. Therefore, we assume an inverse bandwidth of two cycles per cache-line, as seen in previous architecture generations, for all cache-to-cache transfers except L2-L3 on Haswell.

The Haswell architecture offers another architectural change for the ECM model, making predictions more difficult: the uncore frequency, which is the basis for all off-core and inter-core communication. It can differ from the actual clock speed of the CPU. Since the last level cache (L3) is shared among all cores, it is subject to the uncore frequency. The uncore frequency is automatically scaled with the number of active cores and starts off at 2.14 GHz with only one core in use and increases to a peak of 2.98 GHz with three cores. In Table 2.1 we have defined the L2-L3 inverse bandwidth as 2.8 cycles per cache-line in the single-core usecase, which corresponds to 2 cycles per cache-line at a uncore frequency of 2.14 GHz [16].

The Advanced Vector Extensions 2.0 (AVX2) instruction set introduced—among other things—three-operand *fused-multiply-add* (FMA3) support. FMA3 allows the execution of one of the following operations with a single instruction: $a = a \times b + c$, $b = a \times b + c$ or $c = a \times b + c$. The throughput is at 0.5 cycles, since execution ports 0 and 1 each have their own FMA3 execution units. This practically doubles the floating-point peak performance of Haswell CPUs, compared to Ivy Bridge CPUs with the same clock speed. FMA also gives higher precision results [11], since no rounding is done on intermediate results. Although Haswell supports two multiplication or two FMAs, it can only handle one addition per cycle, because only Port 1 supports pure floating-point addition.

2.3. Intel Architecture Code Analyzer

The complexity of understanding scheduling and pipelining of micro-operations is increased by incomplete or vague documentation of these features, but Intel offers a software package called "Intel Architecture Code Analyzer" (IACA) [17], which can predict scheduling and pipelining behavior of modern Intel processors. It does static analysis on a stream of assembly instructions and predicts the maximum throughput of a loop body and the minimum latency of a set of instructions, under the assumption that all data is present in the first level cache. Throughput is useful to examine inner loop bodies, since the code will be run hundreds of times and thus latencies will not matter if sufficient instruction-level-parallelism is available. For codes that do not make good use of the pipeline, latency is the metric of interest. Throughput and latency thus provide best- and worst-case predictions of the execution time, under the assumption that no other bottlenecks apply (such as data transfer latencies from cache misses, NUMA penalties, etc.). All values are returned in cycles.

IACA does not only report the number of cycles but also informs the user of the predicted scheduling including the µops that instructions get split into and the ports they are scheduled to. Micro- and macro-fusion are CPU features to reduce the number of instructions that need to be executed and are also predicted by IACA. An example IACA analysis output can be seen in Appendix A on page 53.

As the name implies, IACA only supports Intel microarchitectures, specifically: from Nehalem to Haswell. Since it operates on binary files and internally disassembles them, there are special marker sequences that need to be inserted into the assembly code around the region of interest before compilation. It is possible to use inline assembly, intrinsics or C macros to mark sections in high-level code, but we have observed that the compiler produces very different code in this case (e.g., without vectorization). Thus it is better and more predictable to manually insert assembly markers.

2.4. LIKWID (Like I Knew What I Am Doing)

The LIKWID toolkit [18] allows the measurement of performance counters during execution of programs, with little to no performance degradation. It takes care of pinning threads to fixed cores, which is especially

useful with OpenMP or other multithreading codes. LIKWID works without any modification of the original code and binary, but it may be used in combination with a marker API, allowing control over the code section to be measured.

The toolkit consists of ten different command line tools. The ones used by our tool are the following:

- `likwid-topology` – show the thread and cache topology
  The relation of hardware threads (individually addressable cores) to physical cores (containing the actual hardware logic) and sockets is reported. This is very helpful for the selection of threads to run on, and it also reports a detailed overview of all cache-levels: giving sizes, associativity, and how many threads share one cache instance. NUMA domains are also reported and give information about the placement of main memory in relation to the hardware threads.

- `likwid-perfctr` – measure hardware performance counters
  This light weight utility gives access to hardware-based performance counters on a wide range of Intel and AMD CPUs and requires no modification of the Linux kernel. A marker API allows selection of a measurement region from within the code, but is not required if the whole execution is to be measured. Pinning must be used, either through `likwid-perfctr` or manually from within the program, because the performance counters relate to specific hardware threads, and measurements can not be mapped to process or operating-system threads. If a measurement is performed on thread 0 and the program is not pinned to thread 0, nothing useful will be measured.

  A number of performance groups are provided, combining the output of several performance counters, because not all counters can be measured at the same time. Also, not all groups are available on all CPU models, either due to restrictions in the performance counter hardware or because of missing documentation.

- `likwid-bench` – benchmarking framework
  Microbenchmarks are vital in assessing the capabilities of modern architectures, due to their complexity deriving from the memory system and instruction level parallelism. This command line tool and framework comes with a dozen ready-to-use assembly benchmarks for streaming kernels. `likwid-bench` takes care of thread spawning, pinning, placement, measurement and reporting in different metrics. The user can chose the desired data set sizes and threads.

The LIKWID toolset is open source software, under the General Public License Version 3, actively developed by Jan Eitzinger and Thomas Röhl at the University of Erlangen. The open source toolset is freely available at [18].

2.5. Performance Modeling

Performance modeling is necessary to understand and to predict performance behavior of code optimizations. The process of modeling is done by reducing the analysis to the most performance relevant hardware and software features. The choice depends on the usecase, i.e., some applications are network bound and others are memory bound or the bottleneck is not the throughput but the latency, which might even depend on mechanical hardware (e.g., backup systems with tape storage). If one could design the perfect performance model for all types of applications, it would be necessary to essentially build or simulate a complete computer system and then end up measuring and not modeling. Thus performance modeling is always a balance between complexity versus accuracy. It is thus driven by an intelligent choice of simplifications.

In the following Sections we will have a look at two models which predict the single-core and chip-level performance of memory, cache and arithmetic bound codes.

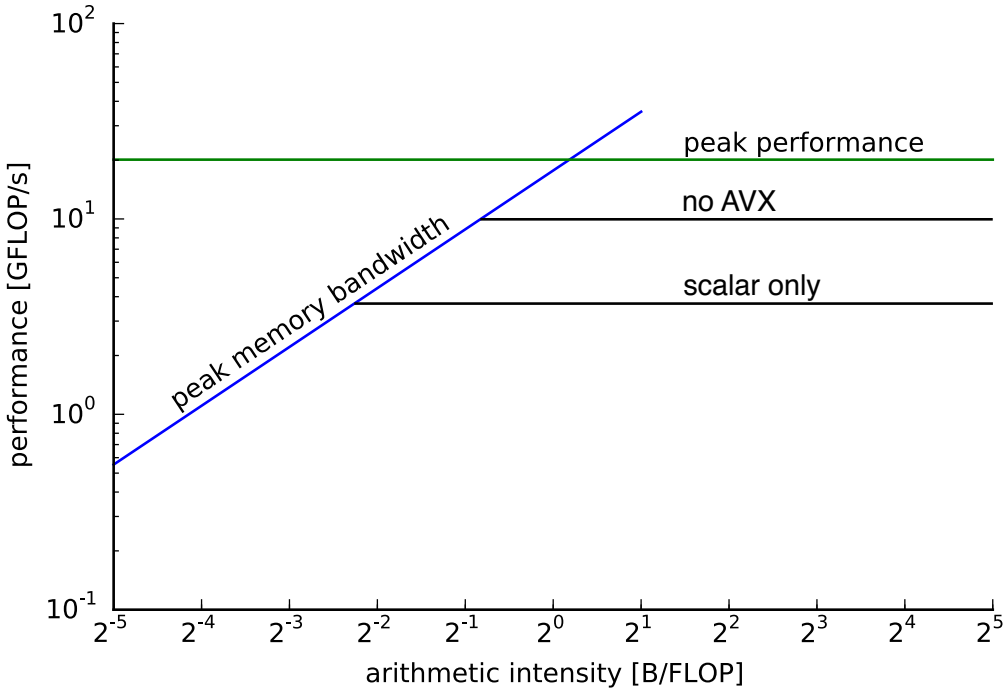


Figure 2.5.: Basic Roofline model for single-core Sandy Bridge Intel Xeon E5-2690 with 10.8 GFLOP/s peak performance and 18.9 GB/s sustained memory bandwidth.

2.5.1. Roofline Model

The Roofline model, published in [19, 20, 21, 22] and named by Williams et al. [1], considers a single metric for the code in question: *operational intensity*, which represents the number of operations per byte of main memory traffic. It is also possible to extend the definition by considering the traffic through all levels in the memory hierarchy, which will give predictions for cache-bound codes. This metric—although simple at first glance—contains a lot of information about the code and its derivation is very often not trivial. Counting the number of operations is usually a simple task, for example by adding up the number of additions and multiplications found in the code section, but main memory traffic means that all traffic solely going to caches is to be ignored when counting, which may be an oversimplification in many cases. Furthermore, calculating the traffic requires a detailed analysis and knowledge of the code and cache architecture. Once the arithmetic intensity is known, metrics about the targeted hardware need to be collected: peak memory bandwidth in bytes per second and peak floating-point performance in FLOPs per second. These represent the upper bounds for memory and performance bound codes. An illustration of the visual representation can be seen in Figure 2.5.

Peak performance and peak memory bandwidth may depend on the CPU features used by the programmer: if not all capabilities are utilized, the upper bound is significantly lower. This can be represented by additional lines, marking areas which are only reachable if certain features are exploited.

To apply this model to a given code, the arithmetic intensity must be computed in order to depict the region of achievable performance in the graphical representation of the Roofline model. If the actual performance can be measured, it is very easy to see if further optimization requires changing the arithmetic intensity (in case of memory bound codes) or make better use of available CPU features, or both. It is also very helpful to get an impression of the absolute theoretically possible performance. Although the graphical representation is usually only used in textbooks and real-world performance engineering is based on the equation introduced shortly.

In mathematical terms, maximum performance P of a code with arithmetic intensity I is defined by the minimum of the CPUs peak performance P_{peak} and the arithmetic intensity times the measured peak bandwidth b_S, yielding the relevant bottleneck:

$$P(I) = \min(P_{\text{peak}}, b_S \times I)$$

This clearly shows the fundamental assumption of the Roofline model: data transfer and computation overlap perfectly.

Peak performance can either be theoretically calculated using the CPU specs or be trimmed towards the code at hand. For example, a Sandy Bridge processor can perform four double precision floating point additions and four multiplications per cycle using the AVX instruction set, with a clock speed of 2.7 GHz., resulting in a theoretical maximum of 21.6 GFLOP/s on a single core. If a code would require only multiplications, peak performance is two times above the actual achievable peak performance. We will call this second definition *application peak performance*.

The peak memory bandwidth is obtained from a stream benchmark, ideally mimicking the access pattern of the modeled application. A typical number for a single core of the processor at hand would be 19 GB/s.

Example

The arithmetic intensity of the 2D-5pt stencil code presented in Listing 1.2 needs to be calculated. First, we need to count the number of floating-point operations (FLOP) required for a single iteration: one multiplication and three additions, thus a total of four FLOP. We can see that one array element is written to (`a[j][i]` or "center") and four are read from (`b[j-1][i]` or "north", `b[j][i-1]` or "west", `b[j][i+1]` or "east", `b[j+1][i]` or "south"), adding up to 40 Byte with double precision floating-point numbers of 128 bit. We add another 8 Bytes to account for write-allocate transfers due to the store misses on `a[]`. We have a total of 48 Bytes that need to be transported between the CPU register and the first level cache (L1) for each iteration. The constant (`0.25`) is assumed to be present in a CPU register and can be ignored, since it does not need loading.

With 48 Bytes to start from, this number can reduce throughout the cache-levels, since some data is read multiple times. In this example the "east" neighbor is used again two iterations later as "west" neighbor. This can be expected to be always the case since a cache does not need to be able to hold a lot of information to cache a few Bytes for two iterations. It gets more tricky with the northern and southern neighbors, since the number of elements that need to be cached depend on the length of the rows or inner dimension. A rule of thumb—for the problem at hand—is the layer-condition [4], which is an easy to evaluate condition:

$$n \cdot N_i \cdot s < \frac{C_k}{2}$$

with n as the number of rows that the stencil spans, N_i as the inner dimension length, element size s and the cache size C_k of cache-level k.

The layer-condition requires that the number of rows used by the stencil completely fits into half the cache size. Half the cache size might be considered pessimistic, but we can not expect that the array in question is the only data residing in the cache. E.g., in the 2D-5pt stencil there are two arrays sharing the cache: `a[][]` and `b[][]`. The given stencil spans three rows ($n = 3$) and is based on 8 Byte elements ($s = 8$ B). We assume an inner dimension N_i of 10^7 elements. The L1 cache has 32 kB, thus we get:

$$3 \cdot 10^7 \cdot 8 \not< \frac{32\,\text{kB}}{2}$$

for L2 we get the following layer-condition:

$$3 \cdot 10^7 \cdot 8 \not< \frac{256\,\text{kB}}{2}$$

and L3 needs to fulfill:

$$3 \cdot 10^7 \cdot 8 < \frac{20\,\text{MB}}{2}$$

The layer-condition therefore postulates that between the main memory and the last level cache only one element of `a[][]` needs to be transferred per stencil update. The write-allocation and updated value of `b[][]` always need to be transferred, since they are not reused. This reduces the total main memory traffic per iteration to 24 Byte. The arithmetic intensity is therefore:

$$I = \frac{4\,\text{FLOP}}{24\,\text{Byte}} = \frac{1\,\text{FLOP}}{6\,\text{Byte}} = 0.167\frac{\text{FLOP}}{\text{Byte}}$$

Using the equation for maximum performance of the Roofline model and taking the single-core data on a Sandy Bridge processor (21.6 GFLOP/s theoretical peak performance, 18.9 GB/s sustained memory bandwidth) we get:

$$P(0.167) = \min(21.6\,\text{GFLOP/s}, 0.167\,\text{FLOP/B} \cdot 18.9\,\text{GB/s}) = 3.16\,\text{GFLOP/s}$$

The application peak performance of this code, with one multiplication and four additions, is at 13.5 GFLOP/s. This would not change the predicted result: the code is therefore memory bound, its performance can not increase beyond that point without increasing the arithmetic intensity.

2.5.2. Execution-Cache-Memory Model

The Execution-Cache-Memory (ECM) model is based on the same bottleneck principle as the Roofline model, but it gives the cache hierarchy more importance and introduces a new—more precise—metric than floating-point operations per second (FLOP/s): cycles per cache-line (cy/CL). Cache-lines refers to the smallest transferable amount of data between main memory and cache-levels (64 Bytes on current Intel processors), and it can also be seen as a workload size. A cache-line workload is work required to update a whole cache-line, varying with the datatype size. For example, a 64 Byte cache-line will result in 8 double precision floating-point numbers and thus in 8 update iterations of the inner kernel loop if the arrays are accessed with stride one.

We will now have a look at the contributions that ECM is taking into account: in-core execution and data transfer time.

In-Core Execution Time (T_{nOL} and T_{OL})

The in-core execution time consists of two parts: one is the time needed to load data from L1 cache into registers and defined as T_{nOL}, and the other is the time for all remaining operations, aggregated into T_{OL}. In both cases, we are only interested in the throughput cycles, due to the nature of loop kernels. The seperation of the in-core execution time into two components has the following reason: research by Stengel et al. [4] has shown that a cycle where a LOAD instruction is retired cannot overlap with any data transfers in the memory hierarchy. This pessimistic assumption is the foundation of the ECM model and the essential distinction from the Roofline model. Thus OL stands for overlapping and nOL for non-overlapping components of the in-core execution time.

Both cycle counts can be gained through an automatic IACA throughput analysis, or by manual assembly code inspection, with respect to the targeted hardware architecture. If a manual inspection is done, the mapping of instructions to µops and pipeline stages, stage delays and port assignment of the hardware need to be known.

Data Transfer Time ($T_{\text{data}} = T_{\text{L1L2}} + T_{\text{L2L3}} + T_{\text{L3MEM}}$)

The data transfer time is split up into inter cache-level transfers and the memory to last cache-level contributions. The first level cache (L1, closest to the registers) has to receive from and transmit to the second

level cache, quantified by the transfer time T_{L1L2}. For stencil codes this is continued down to the lowest level cache, typically L3, where transfers between main memory and this cache-level are covered, e.g., T_{L3MEM}. Transfers between the highest cache-level and registers are covered by T_{nOL}, as mentioned above.

To derive the data transfer times, it is necessary to accurately predict the amount of data transferred between all levels of the memory hierarchy. This can be done using the layer-condition analysis, explained in Section 2.5.1. For pure streaming kernels where there is no temporal locality in data access, the analysis reduces to checking which array resides in which memory hierarchy level. Once the data amounts are known, transfer times between the first level and the last level cache can be calculated based on information found in CPU documentation. For current Intel Xeon CPUs (Sandy Bridge and Ivy Bridge), the inverse bandwidth is given as two cycle per 64 Bytes—one cache-line. The interface between the last level cache and memory is very complex and theoretical bandwidths given in technical documentation do not match up with what is achievable. So, bandwidth measurements need to be used to predict cycles counts at this level (T_{L3MEM}). For most cases simple streaming benchmarks are sufficient, but benchmarks with a matching ratio of load/store streams to main memory may yield more accurate results, because the achievable bandwidth of the memory interface is often influenced by this ratio. An overview of the information required to predict all levels from registers to memory, both in automatic and manual fashion, is presented in Figure 2.6. However, our research has shown that bandwidth measurement based predictions do not yield very accurate predictions on Ivy Bridge and Haswell CPUs. To counteract this effect, we introduced the option to impose *penalty cycles* onto load streams going to memory. The details of how to derive the penalties are given in Section 4. This work-around is not a long term solution, especially since the required penalties seem to grow larger with newer architectures. Eventually, we will need to find an explanation for the behavior and create a suitable model for the inverse memory bandwidth. Recent studies have shown, that it is possible to significantly reduce the penalties by activating the “cluster-on-die” mode and deactivate the downclocking of the uncore frequency in single-core use in the BIOS settings and Intel seems to have approached and mostly eliminated the memory access penalty in the Haswell successor architecture “Broadwell” [23]. Since this information became available very recent and after the evaluation was completed, all Haswell runs presented in Chapter 4 were done with uncore scaling and cluster-on-die disabled.

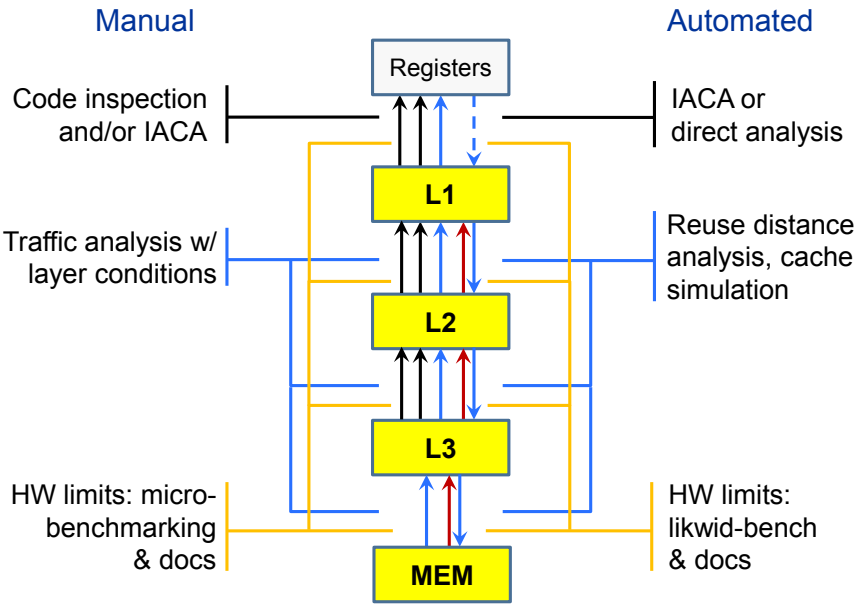


Figure 2.6.: Overview of the information needed for the ECM model cache analysis and their origin in automated and manual analysis. (image: [24])

Single-Core Prediction and Interpretation

To combine all the cycle counts into a single prediction, a general notation has been suggested:

$$\{ T_{\mathrm{OL}} \,||\, T_{\mathrm{nOL}} \mid T_{\mathrm{L1L2}} \mid T_{\mathrm{L2L3}} \mid T_{\mathrm{L3MEM}} \}$$

This can be interpreted in the following way: T_{OL}, to the left of $||$, makes up the in-core execution time and overlaps with the data transfers, to the right of $||$. The data transfers to the right are in the order of increasing distance and latency from the CPU registers. Thus the overall predicted inverse bandwidth is defined as

$$T_{\mathrm{ECM}} = \max(T_{\mathrm{OL}}, T_{\mathrm{nOL}} + T_{\mathrm{data}})$$

Even if a layer-condition is fulfilled at any cache-level, the transfer times still weigh into the total and must be taken into account for an accurate prediction, due to the sequential behavior of data transfers.

There is another notation to present the results which makes predictions of bottlenecks and possible improvements easy to see. By combining the per cache-level predictions

$$T^{\mathrm{L1}}_{\mathrm{ECM}} = max(T_{\mathrm{OL}}, T_{\mathrm{nOL}})$$

$$T^{\mathrm{L2}}_{\mathrm{ECM}} = max(T_{\mathrm{OL}}, T_{\mathrm{nOL}} + T_{\mathrm{L1L2}})$$

$$T^{\mathrm{L3}}_{\mathrm{ECM}} = max(T_{\mathrm{OL}}, T_{\mathrm{nOL}} + T_{\mathrm{L1L2}} + T_{\mathrm{L2L3}})$$

and

$$T^{\mathrm{MEM}}_{\mathrm{ECM}} = max(T_{\mathrm{OL}}, T_{\mathrm{nOL}} + T_{\mathrm{L1L2}} + T_{\mathrm{L2L3}} + T_{\mathrm{L3MEM}})$$

into one term:

$$\{ T^{\mathrm{L1}}_{\mathrm{ECM}} \rceil T^{\mathrm{L2}}_{\mathrm{ECM}} \rceil T^{\mathrm{L3}}_{\mathrm{ECM}} \rceil T^{\mathrm{MEM}}_{\mathrm{ECM}} \}$$

This prediction includes all contributions at each level, e.g., the first numer is applicable if all data is present in the L1 cache and the last number is the prediction if the data does not even fit into the L3 cache. In case of an ASCII representation the backslash (\) symbol is used instead of the `\rceil` ($\rceil$).

Performance is defined by work over time, usually with floating-point-operations or lattice site updates per seconds. To derive performance numbers from the prediction we need to divide the theoretical peak performance, workload (W) times frequency (f), by the predicted cycles counts:

$$P = \frac{W \cdot f}{\{ T^{\mathrm{L1}}_{\mathrm{ECM}} \rceil T^{\mathrm{L2}}_{\mathrm{ECM}} \rceil T^{\mathrm{L3}}_{\mathrm{ECM}} \rceil T^{\mathrm{MEM}}_{\mathrm{ECM}} \}} \,\mathrm{cy} = \{ P^{\mathrm{L1}}_{\mathrm{ECM}} \rceil P^{\mathrm{L2}}_{\mathrm{ECM}} \rceil P^{\mathrm{L3}}_{\mathrm{ECM}} \rceil P^{\mathrm{MEM}}_{\mathrm{ECM}} \}$$

This can also be done for each cache-level, using the "$\rceil$-notation" (or prediction notation) presented above.

Example: 2D-5pt Jacobi Kernel on Sandy Bridge

The two-dimensional 5-point Jacobi kernel we have seen before has the following access pattern in the innermost loop: `b[j][i] = (a[j][i-1] + a[j][i+1] + a[j-1][i] + a[j+1][i])`. If we assume 64 bit double precision floating point numbers and 64 Byte cache-line length, we get 8 elements and 8 iterations to update one complete cache-line. The arrays `a[][]` and `b[][]` both have the same dimensions, but only the inner (second) dimension is relevant here (since the layer-condition is independent of the outer dimension). We choose it to be 5000 elements long.

First we need to estimate the cycles required to load data from L1 (T_{nOL}). This could be done by IACA (see Appendix A), but can also be done by hand, since we know that we can load two times 128 bit per cycle (see Table 2.1). For each update of four doubles with AVX, we need to load four times 256 bit (four north, south east and west elements). This adds up to $16 \cdot 64$ bit per half cache-line and four cycles. This results in a total of 8 cycles for one cache-line worth of work:

$$T_{\mathrm{nOL}} = 8\,\mathrm{cy}$$

For the same workload, 24 individual adds and 8 multiplies are necessary, which can be reduced using the AVX instruction set to 6 add and 2 multiply instructions. Multiply and add can be scheduled in parallel and have a throughput of 1 cycle each. This leads to an overlapping time of six cycles:

$$T_{\mathrm{OL}} = 6\,\mathrm{cy}$$

The in-core execution times are dominated by the LOAD pipeline and a more exact and detailed analysis can be gained using IACA (see Appendix A). Next comes the data transfer analysis, which can be done manually by employing the layer-condition. In case of the Sandy Bridge processor Intel Xeon E5-2680 (see Table 2.1), there are three cache-levels with 32 kB (per-core) L1, 256 kB (per-core) L2 and 20 MB (shared) L3 cache.

In case of 5000 elements per row (inner dimension), the layer-condition is met in the L2 cache:

$$3 \cdot 5000 \cdot 8\,\mathrm{B} < \frac{256\,\mathrm{kB}}{2}$$

Thus only 64 bit (one double) need to be loaded from L3 and main memory (plus 64 bit from `b[][]`, because of write allocate) in addition to 64 bit for storing `b[][]` per cache-line, since L2 is caching everything else. Between L1 and L2, three (plus two) floats need to be loaded, since the second float accessed on the center line is already present in the cache. A visual representation of the cache usage can be found in Figure 3.2c on page 31. We want to predict the number of cycles, so we need to combine the transfer rates with the amount between L1 and L3:

$$T_{\mathrm{L1L2}} = 8\frac{\text{iteration}}{\text{cache-line}} \cdot \left(\underbrace{3\,\frac{\text{double}}{\text{iteration}}}_{\text{load from a[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{load from b[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{store to b[][]}} \right) \cdot 8\,\frac{\text{Byte}}{\text{double}} \cdot \frac{2\,\frac{\text{cycle}}{\text{iteration}}}{64\,\text{Byte}} = 10\,\frac{\text{cycle}}{\text{cache-line}}$$

$$T_{\mathrm{L2L3}} = 8\frac{\text{iteration}}{\text{cache-line}} \cdot \left(\underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{load from a[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{load from b[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{store to b[][]}} \right) \cdot 8\,\frac{\text{Byte}}{\text{double}} \cdot \frac{2\,\frac{\text{cycle}}{\text{iteration}}}{64\,\text{Byte}} = 6\,\frac{\text{cycle}}{\text{cache-line}}$$

$$T_{\mathrm{L3MEM}} = 8\frac{\text{iteration}}{\text{cache-line}} \cdot \left(\underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{load from a[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{load from b[][]}} + \underbrace{1\,\frac{\text{double}}{\text{iteration}}}_{\text{store to b[][]}} \right) \cdot 8\,\frac{\text{Byte}}{\text{double}} \cdot \frac{2.7\,\frac{\text{Gcycle}}{\text{second}}}{40\,\frac{\text{GByte}}{\text{second}}} \approx 13\,\frac{\text{cycle}}{\text{cache-line}}$$

Although these calculations seem complicated, it is easy to reproduce if all metrics are based on cache-lines. Combining the prediction with the fulfilled layer-condition in L2, we get:

$$\{\, 6 \,||\, 8 \,|\, 10 \,|\, 6 \,|\, 13 \,\}\ \text{cycles}$$

Figure 2.7 shows the individual time contributions to the total prediction and how they relate to one another (overlapping or non-overlapping).

To give a better representation of the total cycles required if data is contained in a certain cache-level, we use the prediction-notation:

$$\{\, 8 \rceil\, 18 \rceil\, 24 \rceil\, 37 \,\}\ \text{cycles}$$

and can be represented in a performance metric, e.g., giga FLOPs per second:

$$\{\, 2.7 \rceil\, 1.2 \rceil\, 0.9 \rceil\, 0.6 \,\}\ \text{GFLOP/s}$$

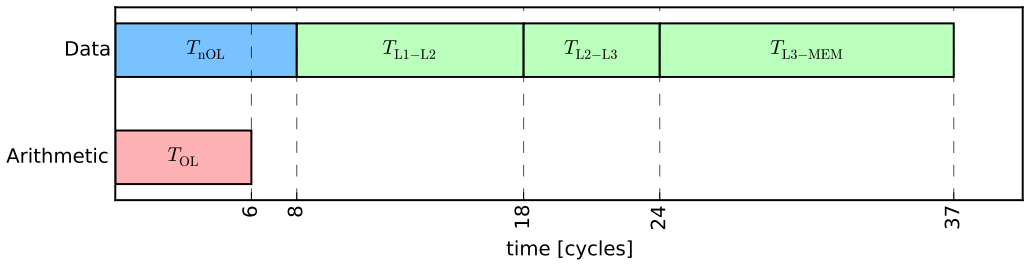


Figure 2.7.: The ECM Prediction for 2D-5pt Jacobi Kernel on Sandy Bridge E5-2680, abovet are the data transfer contributions and bellow the overlapping portion from in-core latencies.

Multi-Core Prediction

An assumption of the ECM model is, that the single-core predictions can be used to predict the multi-core behavior by scaling up until a bottleneck is hit. The bottleneck will typically be the transfers between the last level cache and main memory, since that is the only non-scaling bandwidth on the architectures examined here[2]. The performance scaling over n cores is described as

$$P(n) = \min\left(nP^{\mathrm{MEM}}_{\mathrm{ECM}}, b_S \cdot I\right)$$

with the memory bandwidth b_s, code intensity I and their product being Roofline's upper memory-bound performance. Thus, to get the number of cores a code scales to we can use the following saturation formula from [4]:

$$n_s = \left\lceil \frac{b_s \cdot I}{P^{\mathrm{MEM}}_{\mathrm{ECM}}} \right\rceil = \left\lceil \frac{T^{\mathrm{mem}}_{\mathrm{ECM}}}{T_{\mathrm{L3MEM}}} \right\rceil$$

with

$$T^{\mathrm{mem}}_{\mathrm{ECM}} = T_{\mathrm{nOL}} + T_{\mathrm{L1L2}} + T_{\mathrm{L2L3}} + T_{\mathrm{L3MEM}}$$

which is the number of cycles per cache-line if all data resides in memory.

In addition to the bandwidth, the cache size of the L3 cache does not scale either, since it is shared among all cores. This can have the effect that the L3 cache-size per core is reduced, unless the same data regions are accessed by multiple cores at the same time. We will ignore this for now and explain our simplified resolution of this case in Section 3.

[2] On the Intel Westmere CPU, the L3 cache was a bandwidth bottleneck since it could only deliver 32 Bytes/cy in total, not scaling with the number of cores. In cases where the L3 cache traffic is much larger than the memory traffic (e.g., if many layers need to be "caught" at L3), the L3 cache on Westmere may be the relevant bottleneck.

3 IMPLEMENTATION

The developed toolkit is called *kerncraft*. The source code and example inputs can be found on github[1] and are licensed under the GNU Affero General Public License version 3 (AGPLv3) [25]. kerncraft is also available as a python package, listed on the Python Package Index (pypi) [26].

The command `kerncraft` allows access to the analysis with all implemented performance models. For any analysis a model (e.g., ECM), a machine description file and kernel code, as well as possible constants (e.g., N=64) need to be provided. After completion, the computed performance bounds are reported to the user through standard out. An example run can be seen in Listing 3.1. Information that was gathered during the static analysis can also be reported using the verbose flag `-v`. The usage and the most common command line arguments are explained in Section 3.4.

```
$ kerncraft -m phinally.yaml 2d-5pt.c -p ECM -D N 5000 -D M 500
================================================================================
                                kernels/2d-5pt.c
================================================================================
{ 9.0 || 8.0 | 10 | 6 | 12.7433628319 } = 36.74 cy/CL
{ 9.0 \ 18.00 \ 24.00 \ 36.74 } cy/CL
saturating at 3 cores
$
```

Listing 3.1: Example run of kerncraft with ECM model and 2D-5pt Jacobi kernel on a 5000 by 500 element matrix.

3.1. Structure

Internally kerncraft consists of the following components: the command line interface (CLI), the kernel code analysis, the performance models and the machine model. The CLI is provided by the `kerncraft.py` file, the main entry point for users. All kernel codes go through the analysis by the module found in `kernel.py`, collecting all static information that needs to be extracted. Each performance model is a submodule found

[1] `https://github.com/cod3monk/kerncraft`

in the `models` directory, which currently are: `ECM`, `ECMData`, `ECMCPU`, `Roofline`, `RooflineIACA` and `Benchmark`. The machine model is found in `machinemodel.py` and parses machine description files.

Additional helper scripts exist to collect and manage data that is generated: `iaca_marker.py` helps adding IACA markers to the assembly code, `likwid_bench_auto.py` collects all possible and necessary information on the hardware architecture using `likwid-bench` and `likwid-topology`. `picklemerge.py` allows to merge gathered results from kerncraft for later batch analysis.

A general overview of the internal workings behind kerncraft is depicted in Figure 3.1.

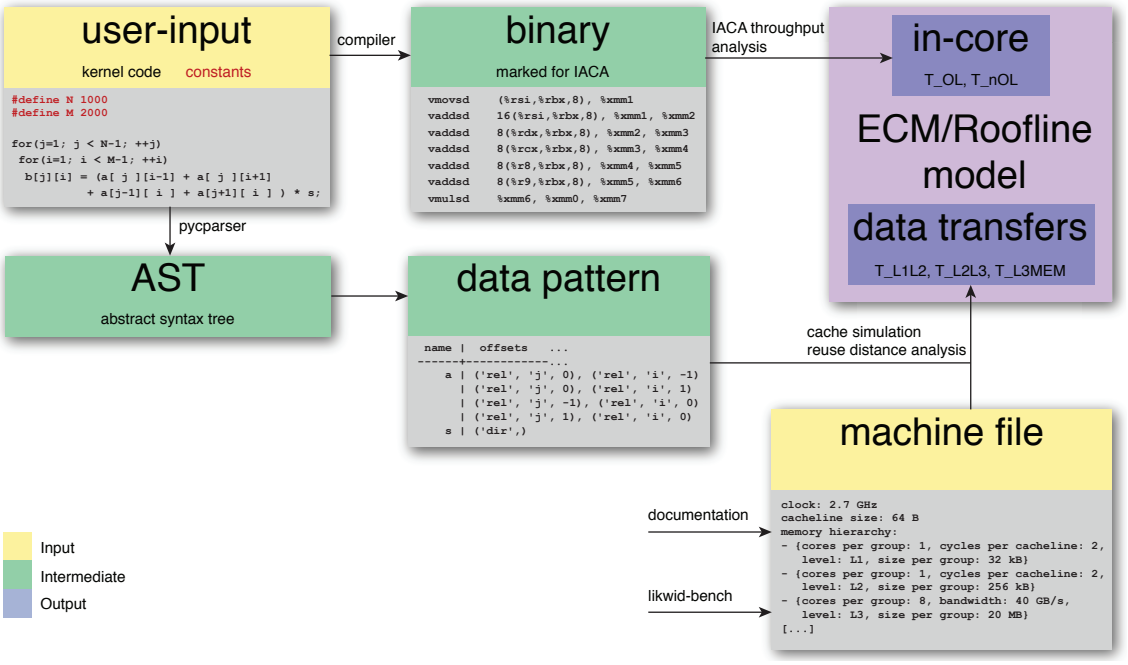


Figure 3.1.: Overview of the kerncraft toolkit analysis

3.2. Kernel Code Analysis

All information that can be gathered from the source and its compiled binary is handled in the `kernel` module. It can be split in two areas: the static source code analysis and the IACA based analysis. Support functions to parse, generate, assemble and compile the code are also part of this module. In the following sections we will have a detailed look at an example analysis of a 2D, 5 point Jacobi stencil code, explain the analysis and what information is gathered and how it is extracted.

3.2.1. Kernel Code

The basis for the kernel code syntax is the ISO C Standard 1999 [27] (C99), but some restrictions are necessary to make analysis feasible. From now on, *constant* will refer to a value passed to the kernel from the command line and it is used to define array dimensions. An example kernel code can be seen in Listing 3.2.

```
double a[M][N];
double b[M][N];
double s;

for(int j=1; j<M-1; ++j)
    for(int i=1; i<N-1; ++i)
        b[j][i] = ( a[j][i-1] + a[j][i+1]
                  + a[j-1][i] + a[j+1][i]) * s;

}}}
```

Listing 3.2: 2D-5pt Jacobi kernel code

All kernel codes must fulfill the following restrictions:

- Statements need to be multidimensional arrays or scalar declarations
- Only datatypes `double` and `int` are supported, due to their relevance in scientific codes. Support for other constant sized datatypes, such as `float`, can be added easily.
- Array declarations may use fixed sizes or constants, with an optional addition or subtraction of an integer (e.g., `double u1[M+3][N-2][23]`, but not `double u[M*N]`).
- The declaration statements must be followed by exactly one for-loop. This loop may either contain further for-loops or multiple assignment statements.
- It is not allowed to mix loops and assignment statements within a loop body. Only the innermost loop may contain assignment statements.
- Assignment statements in the innermost for-loop must comply with the following: valid operands are scalars, integer, float or double values and array references.
- Array references must either use index variables from for-loops, with optional addition or subtraction, constant or fixed values.
- All for-loops must use a declaration as initial statement and an increment or a decrement assignment operation as the next statement (e.g., `i++`, `i -= 2`). The condition statement must compare the same index variable used in the initial and next statements with a fixed integer value or a constant with an optional addition or subtraction of an integer using the less-than operator (e.g., `i < N+2`).
- Function calls and the use of pointers is not allowed anywhere in the kernel code.
- Write access to any data is assumed to use “normal” STORE instructions (e.g., not non-temporal stores).

These restrictions might seem very limiting, but all stencil codes and many other loop based kernels can be represented easily in such a manner, especially if the original code is based on C or C++. Some of these restrictions are even taught as good practice in basic high performance computing classes [28]. For example: by keeping if statements out of the loop constructs—especially the inner loops—performance gains will often be significant.

3.2.2. Code Parsing and Static Analysis

We use the pycparser [29] python library for parsing of kernel codes, which supports the C99 standard. The resulting abstract syntax tree (AST) is then checked for the restrictions mentioned Section 3.2.1. From the static analysis we need to learn about three different things:

- for-loop stack
- data sources and destinations
- floating point operations used

The loop stack contains information about all for-loops present in the code: their order, index variable name, start value, end value and step size. This gives us the number of loop iterations that will be performed and allows us to figure out the order of accesses to arrays. See Table 3.1a for the gathered information from Listing 3.2.

Data sources and destinations are retrieved from statements in the inner for-loop. Any data access can be either direct (e.g., to a scalar or an array with constant or fixed index) or relative with an optional offset. Multidimensional arrays have relative or direct access on each dimension, for instance `xy[0][j][i+1]` is a direct access on the first dimension, than a relative access on the second dimension and another relative relative access on the third dimension with an offset of +1. See Table 3.1b and 3.1c for the data access analysis of the kernel code found in Listing 3.2.

Floating point operation (FLOP) counts are simply the number of operations found in the inner statements, possible compiler optimizations such as common subexpression elimination or compile-time evaluation are being ignored here. The gathered counts are only used by the Roofline model and not for the ECM prediction. The counts for Listing 3.2 are found in Table 3.1d.

index variable	start	end	step size
j	1	499	+1
i	1	4999	+1

(a) Loop stack

variable	1st dimension	2nd dimension
a	relative j	relative $i-1$
	relative j	relative $i+1$
	relative $j-1$	relative i
	relative $j+1$	relative i
s	direct	

(b) Data sources

variable	1st dimension	2nd dimension
b	relative j	relative i

(c) Data destinations

operation	count
+	3
*	1

(d) Floating point operations

Table 3.1.: Information gathered from 2D-5pt Jacobi kernel in Listing 3.2 with constants N=5000 and M=500

3.2.3. IACA Analysis

The Intel Architecture Code Analyzer (IACA) is a static code analysis tool used to predict throughput and latency of code sections on Intel microarchitectures. More information on the software can be found in Section 2.3 and at [17]. We use IACA to analyze the in-core performance of the kernel code, because this tool is freely available and contains information which is not easily extractable from Intel's public documentation. The drawback is that older CPU microarchitectures and architectures by other vendors are not supported.

For IACA, the code needs to be transformed into a compilable version and the relevant section marked using special assembly instructions. To do so, the kernel code is transformed by first inserting it into a main function. For each constant, code is inserted to take an argument from the command line and saved as a constant integer variable. Afterwards, all array declarations are replaced by pointer declarations with memory allocation using `_mm_malloc`, in order to produce 32 byte aligned and arbitrary sized arrays. It is necessary to do so, because otherwise, the allocation would take place in the stack section of memory which is typically limited to 8 MB. The loop stack is left untouched, but the inner statements are changed if they contain multidimensional array references. The multi-dimensional indexes need to be replaced by single-dimension index arithmetic to map all values to a continuos one dimensional array. In the end an external dummy function call is inserted and is passed a reference to the arrays. This will make sure that the compiler cannot over-optimize the code and partially or completely remove statements.

The throughput analysis of the whole binary is not relevant, because we are only interested in the throughput of the innermost loop. To restrict the analyzed section, IACA needs to find special assembly instruc-

tions that can either be inserted via intrinsics in the C code or by manually modifying the assembly produced by the compiler. Intrinsics have the disadvantage that they can throw the compiler off track, keeping it from vectorizing the loop code. Since we are interested in compiler optimized code, this approach is not suitable. Finding the correct block in the assembly code is not too difficult and is done by selecting the block with the most vector instructions. In case this fails, the user can always overwrite the automatic behavior using the `--asm-block` command line argument.

The unrolling factor is extracted from the marked section by analyzing the loop counter increments used at the end of the block. This is needed to allow interpretation of the throughput analysis results from IACA. For example, if a loop is 4-times unrolled and IACA predicts a throughput of 32 cycles, then one single loop iteration and update step takes 8 cycles. This analysis is valid for scalar and vectorized loops, regardless of the processor features used.

3.3. Performance Models

We have implemented three performance models, which are based on the previously mentioned code analyses: Execution Cache Memory (ECM), Roofline and Benchmark. All models can operate on all supported codes. Both the ECM and Roofline analysis can be executed on any machine and their output does not depend on the hardware it is run on, since the code is never actually run. The Benchmark model is special, since it actually executes the compiled kernel code using `likwid-perfctr` to measure the performance, rather than predicting it. All models are python classes that need to be implemented according to a kerncraft specific interface: Each model class must offer a constructor accepting the kernel object, a machine description object, the command line arguments and the command line parser object. After creation, two member functions are called without any arguments: `analyze()`, which is expected to do all the analysis work and store the results in an instance variable by the name of `results`, and `report()`, which is supposed to print an analysis report to stdout based on the information found in the `results` variable.

The central part of kerncraft, and one crucial ingredient in analytical performance modeling, is the cache reuse analysis in `ECMData` and `Roofline`. This is required to predict the origin of data during accesses. There are certain assumptions towards the hardware and software side that allow the generic analysis algorithm to produce useful results:

- *Perfect least-recently-used (LRU) cache replacement strategy on all cache levels*
  Although Intel CPUs formerly used a pseudo LRU (PLRU) strategy as seen in [15], and most likely still do, they try to mimic LRU and thus this assumption is acceptable.
- *Small warm-up effects*
  When a loop runs for the first time, the cache is not filled with the relevant data as assumed by the prediction model. This will be negligible, as long as the loops are iterated often enough.
- *Proportionally short peel and remainder loops*
  Compilers have to create special loops in the code to handle remaining loop iterations from unrolling (remainder loops) and from establishing alignment constraints (peeled iterations). As long as the total number of iterations is high, remainder and peel loops can be ignored in the prediction.
- *Perfect prefetching*
  The ECM and Roofline models expect prefetching to hide all latencies in the memory hierarchy, so that the transfer time for any amount of data across a specific data path is just the ratio of data volume and asymptotic bandwidth. Here we will rely on hardware prefetching of the Intel Xeon processors.
- *No resource sharing due to multiprocessing*
  We assume that all available resources are solely dedicated to the execution of the loop kernel. This means that cache sizes, bandwidths and cycles are not shared, which is never true, because the operating system and other background daemons will take their share, but it is a good approx-

imation on single-user, dedicated compute nodes. In case of multi-core prediction, it is possible to specifically share L3 cache-size and memory bandwidth.

- *Scalar variables are kept in registers*
  All scalar variables used in kernel codes are ignored in the data access, since they are assumed to be present in registers during loop iterations.
- *Fully associative caches*
  Since real memory locations are unknown during static analysis, exact prediction of collisions is not possible.
- *Fully inclusive cache hierarchy*
  We expect that all caches keep a copy of the information passed onto the next (smaller) cache level. This is the case with the investigated processors.

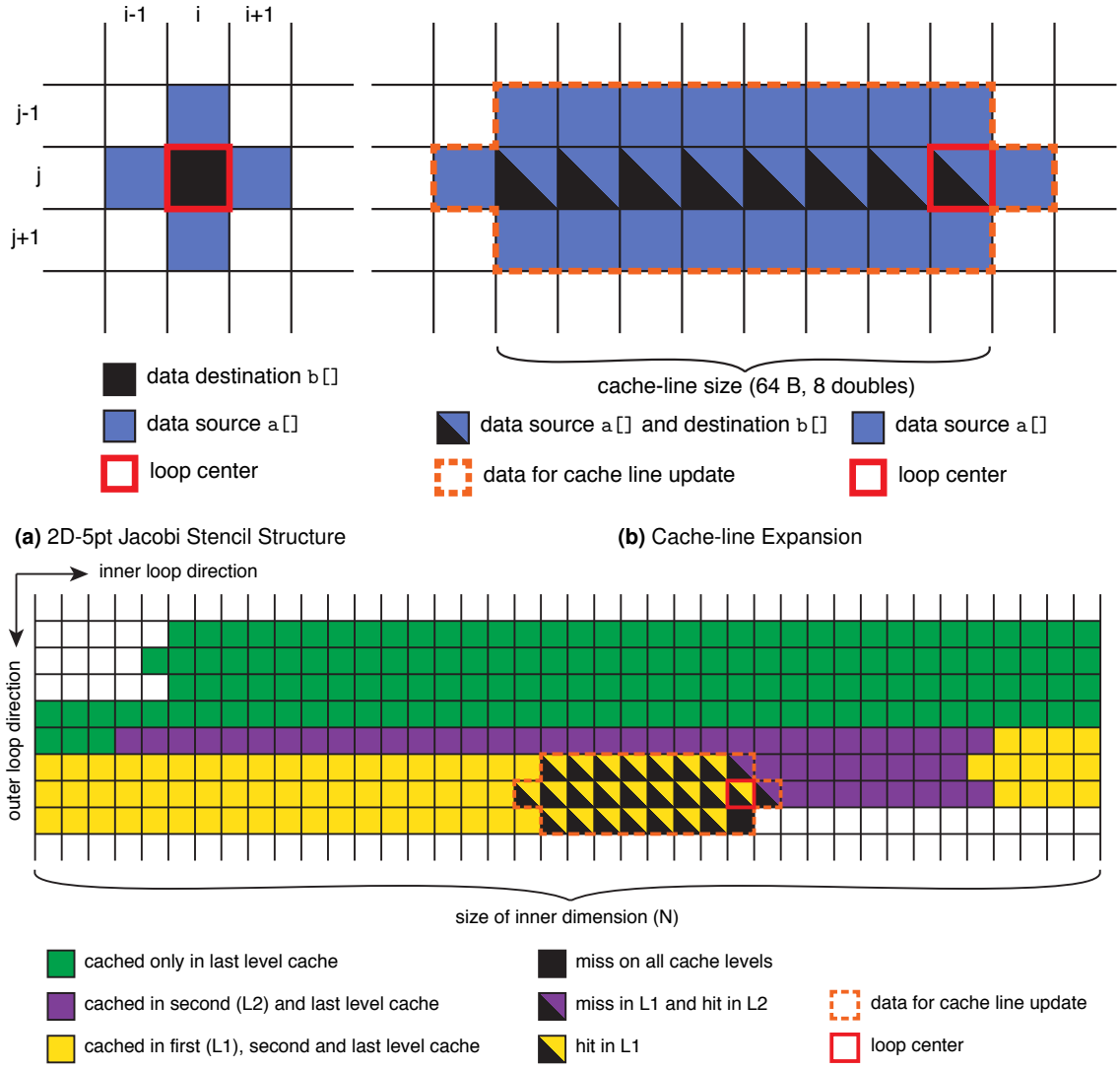
Some of these restrictions or simplifications towards the cache implementation in the CPU may be lifted by combining kerncraft with a more complex cache simulation, as planned for future work.

3.3.1. Cache Prediction Algorithm

Each cache level is inspected independently, although only misses in lower levels (closer to registers) are passed to the next higher level. The analysis yields the same result if all caches are hit with all requests, simplifying the implementation. We start off with all data requests from the loop iteration *i*, *j* in the 2D-5pt Jacobi kernel code from Listing 3.2 shown in Table 3.1b, respectively 3.1c: `a[j-1][i]`, `a[j][i-1]`, `b[j][i]`, `a[j][i+1]`, `a[j+1][i]`. A visual representation can be seen in Figure 3.2a. The grid lines depict the underlying matrix cells. Unfortunately, this notation does not offer any information about the actual location of the values in memory, so we change to a 1D-notation: `a[(j-1)*N+i]`, `a[j*N+i-1]`, `b[j*N+i]`, `a[j*N+i+1]`, `a[(j+1)*N+i]`. The mapping from 2D to 1D emphasizes the streaming nature of the data accesses, but without *N*, *i* and *j*, it is still not sufficient. For *N* we will need to choose a number, for this example we will use $N = 40$. *i* and *j* we will keep abstract and say that the loop center is at a relative offset of 0. If we plug this back into the 1D offsets, it leaves us with the following relative offsets: -40, -1, 0, +1, +40.

Since in the modeling works with cycles per cache-line, we need to look at the data needed for a workload of one cache-line. To do so, we take the loop center and move it as many iterations back as elements fit into one cache-line, which is 8 doubles, thus 8 iterations. In this case it is easy, since the step size of the inner loop is +1, the cache-line expansion can be seen graphically in Figure 3.2b. This gives us all data accesses needed to take into consideration for one cache-line worth of workload. We can already see that accesses on the center line are overlapping and will be cache hits. They can be removed, since it can be assumed that everything for one cache-line will easily fit into the lowest cache level (32 kB on current Intel Xeon processors). In this pattern, all offsets are treated as misses, since they need to be loaded to fulfill the cacheline sized workload. Cells marked black are store accesses to `b[]` and cells marked blue are load accesses from `a[]`.

We now start adding iterations in the same fashion as we have seen, until the cache size is exceeded. After each addition of new offsets, they are checked for overlaps with the original set of accesses. If there is an overlap this means a cache hit. Once the cache is full, it is easy to see that all offsets from the original set which were not turned into hits, must be misses and thus contribute to inter-cache-level or memory traffic. See Figure 3.2c. for a complete illustration of all three cache levels. Note that, if there are many hits in the first level cache (L1), they are also hits in the second and last level, but since only misses lead to utilization of bandwidth, they are the information we seek. We can find one miss going all the way to memory, because the complete dataset does not fit into memory, it is located bellow the loop center and has the relative offset of -40. Also, there are two misses on L1 to the right of the loop center and above it, at offsets -1 and +40, but they are hits on L2. Overall we can say that the layer-condition is fulfilled in the L2 cache level.



(a) 2D-5pt Jacobi Stencil Structure

(b) Cache-line Expansion

(c) Cache Usage Prediction with N=50

Figure 3.2.: Cache Prediction Algorithm

The actual communication between caches and memory is done in memory blocks or cache-lines (synonyms), the number of cache-lines therefore needs to be calculated from the offsets. We do not know any real memory address, so we arbitrarily decided that the first cache-line starts at offset 0 inclusively.

In order to take care of write-allocates, all writes offsets are also treated like reads. It is currently not possible to handle write-through, because it was not needed with the investigated codes. All write offsets are added to an evict list and no caching is tracked on this, meaning that all writes are immediately evicted as long as they go to offsets in arrays.

3.3.2. ECM

The ECM model is split into two submodels, one being the data access and cache analysis, also known as `ECMData`, and the other being the in-core analysis performed by IACA, called `ECMCPU`. The model `ECM` combines both submodels and gives an overall prediction. The reson for the splitting is that the IACA tool may not be available on all systems, and it does certainly not support microarchitectures other than Intel's. In such cases one may still use the `ECMData` model to get the data traffic analysis.

Multi-Core Prediction

Based on the bottleneck analysis mentioned in Section 2.5.2, `kerncraft` always reports the number of cores needed for saturation (n_s). By default, the only shared resource considered is the memory bandwidth. To take the shared L3 cache size into account, it is necessary to use the command-line argument `--cores` and pass it the number of cores to scale to. This will effectively reduce the available L3 size for the cache usage prediction. Since all architectures mentioned in Table 2.1 have a shared L3 cache, only that cache size is considered.

3.3.3. Roofline

The Roofline analysis uses the same algorithm as the ECM analysis to extract data access patterns between the first level cache and all other levels in the memory hierarchy. The interpretation of the data differs in the following way. For each cache level, the arithmetic intensity is calculated by dividing the floating point operations by the number of bytes transferred to and from that cache level. The arithmetic intensity is then used in combination with benchmark results in the machine description to predict a maximum performance that can be expected at the given level. The machine description contains results from multiple bandwidth benchmark kernels (e.g., load, copy, triad), and here we selected the one matching the observed data pattern best, in terms of read vs. write streams. Each memory or cache level and the CPU itself are considered a potential bottleneck. After upper bounds on all levels have been computed, the slowest one is chosen and returned as the Roofline prediction.

The CPU bottleneck can be calculated in two ways: either using IACA (`RooflineIACA` model) or by taking the theoretical peak performance of the CPU (`Roofline` model). If the theoretical total peak is considered, the first level cache bandwidth to CPU register performance needs to be viewed as another cache level. The peak performance is usually an over estimation, because it assumes the perfect mix of operations for the given CPU and a more suitable model will replace this in future work. On the other hand, the maximum arithmetic CPU performance and first level cache to register performance can be obtained more accurately by an IACA throughput analysis, as described in Section 2.3.

3.3.4. Benchmark

Unlike the previous models, using the command line argument `--pmodel Benchmark` does not predict performance, but measures it. Therefore this model must also be executed on the same machine as described in the machine file. Although most information from the machine file will not be necessary, it is

used to determine the appropriate compiler flags, cache-line size and clock speed. For the measurement results to be generated correctly, the user must set the CPU frequency to the clock speed defined in the machine file.

In order to allow execution of the kernel code, it needs to be transformed in a similar manner as described in Section 3.2.3. Here, we insert one extra for-loop wrapping all kernel loops to prolong the execution time to at least half a second, and calls to the LIKWID marker application programming interface (API) are inserted before and after the outer for-loop, for precise measurements.

After successful transformation and compilation, the binary is executed with `likwid-perfctr`, LIKWID's performance counter and pinning command. Pinning to the first core is therefore ensured and runtime can be measured alongside memory bandwidth and data volume, cycles per instruction and other metrics. The runtime is used to calculate the number of cycles per work unit.

3.4. Usage

Kerncraft consists of multiple command line tools, the main one being the `kerncraft` command. Other commands are `iaca_marker`, a tool to mark assembly code blocks for later IACA analysis without compromising on compiler optimizations and `likwid_bench_auto`, which allows the user to collect information about the machine and generate a machine description file. `picklemerge` merges two or more Python pickled dictionaries recursively, specifically the results files created by kerncraft for later analysis. In this Section on we will have a look at `kerncraft`'s command line interface and how to use it.

To use `kerncraft` one has to provide two files: one machine description and a kernel code file. The machine description files are in the Yet Another Markup Language (YAML) format [30] and can be generated using the `likwid_bench_auto` command. The generated file contains only partial information about the machine, since some data (e.g., cycles per cache-line transfer) can not be extracted, but need to be manually extracted from developer manuals and documentation published by vendors. All missing values are marked by the placeholder string `REQUIRED_INFORMATION` and need to be filled in before the file can be used with `kerncraft`. The kernel code syntax and requirements are described in detail in Section 3.2.1. Example machine description files can be found in the kerncraft repository for the compute nodes at the RRZE HPC center described in Table 2.1, as well as the kernel codes described in Sections 4.1 and 4.2. The generation, interpretation and format of the machine description is explained in Appendix B.

The command line interface (CLI) accepts the following format:

```
kerncraft --machine MACHINE --pmodel PMODEL [-D KEY VALUE] [--verbose] [--asm-block
     BLOCK] KERNELFILE [KERNELFILE ...]
```

We will have a look at the most commonly used options, but more information about the arguments can be found by calling `kerncraft --help`.

- `MACHINE` needs to be replaced with the location of the machine file and
- `KERNELFILE` with the location of the kernel code, where as multiple can be passed for batch processing.
- Depending on the kernel code, constants must be passed via `-D KEY VALUE` (e.g., `-D N 100` will set constant `N` to 100) and multiple constants can be passed by repeating the `-D KEY VALUE` argument.
- The `--asm-block BLOCK` parameter influences only models that are based on an IACA analysis: it can either be the index of an assembly block to mark for IACA or `manual` if the selection should be done interactively. If the parameter is not given, it defaults to an automatic selection based on statistics reported with `manual`. See Appendix C for information on usage and default behavior.
- `PMODEL` must be one of `{ECM,ECMData,ECMCPU,Roofline,RooflineIACA,Benchmark}`. Information about the different performance models can be found in the preceding Sections.

- During investigation of a kernel code, it is very helpful to make use of the `--verbose` flag (short `-v`), which can be passed one or more times to increase verbosity of the output reporting.

An example run can be seen in Listing 3.1. The used kernel file (`2d-5pt.c`) contains exactly the code in Listing 3.2 and this specific machine description file (`phinally.yaml`) is presented and explained in Appendix B.

4 EVALUATION

We will have a look at ECM predictions from kerncraft versus performance measurements on three microarchitectures with six stencil and six stream kernel codes. The system and microarchitecture specifications used can be found in Table 2.1. In the following sections, we will present the kernel codes used for evaluation, followed by the comparison of prediction and measurement results. The software stack used on the three architectures consists of a Linux operating system with the following software packages directly used for evaluation: Intel Automatic Code Analyzer (IACA), Intel Compiler (ICC), LIKWID, Python and of course the kerncraft toolkit. An overview of the versions used on each system is given in Table 4.1.

Software package	Version on phinally (Sandy Bridge)	Version on ivyep1 (Ivy Bridge)	Version on hasep1 (Haswell)
Intel Automatic Code Analyzer (IACA)	2.1 (32 bit)	2.1 (32 bit)	2.1 (32 bit)
Intel Compiler (ICC)	13.1.3 20130607	13.1.3 20130607	13.1.3 20130607
kerncraft	284979 (git)	284979 (git)	284979 (git)
LIKWID	3.1	3.1	4.0
Python	2.7.3	2.7.3	2.7.6

Table 4.1.: Utilized Software Stack

As explained in Section 2.5.2, we had to find a penalty factor for Ivy Bridge and Haswell L3-Memory access. To do so, we compared the predicted cycles to the measured cycles per cacheline in the worst-case, where no layer-condition is fulfilled and all L1 cache misses are forwarded to main memory. Table 4.2 shows the deviations of all predictions to benchmarks and how they relates to the number of read streams. We can see that phinally (Sandy Bridge) has little differences, ivyep1 (Ivy Bridge) has some and hasep1 (Haswell) has most. From the data, we selected the penalty for Ivy Bridge to be 3 cycles per read stream going to memory and 4 cycles for Haswell, which is part of the machine description described in Appendix B. All following results already contain the penalty in the T_{L3MEM} component.

System	Kernel	read/write streams	prediction [cy/CL]	benchmark [cy/CL]	difference [cy/CL]	deviation[a] [%]	diff. per read str.
phinally	2d-5pt	4/1	49.9	52.9	3.0	6.0	0.7
	3D long range	11/1	185.6	199.2	13.6	7.3	1.2
	3D-7pt	4/1	57.9	58.7	0.8	1.4	0.2
	triad	4/1	47.9	55.5	7.5	15.9	1.9
	scale	2/1	26.7	29.9	3.2	12.0	1.6
	add	3/1	37.0	42.9	5.9	15.9	2.0
	scalar product	2/0	19.8	24.8	5.0	25.3	2.5
	3D-27pt	4/1	116.4	113.1	-3.3	-2.8	-0.8
	vector sum	1/0	9.9	14.2	4.3	43.4	4.3
	DAXPY	2/1	28.7	31.1	2.3	8.4	1.2
	uxx stencil	9/1	132.4	135.1	2.7	2.0	0.3
	1D-3pt	2/1	30.7	42.6	11.8	38.8	5.9
ivyep1	2D-5pt	4/1	48.0	58.4	10.4	21.7	2.6
	3D long range	11/1	181.0	205.9	24.9	13.8	2.3
	3D-7pt	4/1	56.0	60.3	4.3	7.7	1.1
	triad	4/1	46.0	64.8	18.8	40.9	4.7
	scale	2/1	26.0	31.3	5.3	20.4	2.7
	add	3/1	36.0	46.0	10.0	27.8	3.3
	scalar product	2/0	20.0	23.8	3.8	19.0	1.9
	3D-27pt	4/1	114.5	97.3	-17.2	-15.0	-4.3
	vector sum	1/0	10.0	12.9	2.9	29.0	2.9
	DAXPY	2/1	28.0	32.8	4.8	17.1	2.4
	1D-3pt	2/1	30.0	40.5	10.5	35.0	5.2
	uxx stencil	9/1	128.5	147.7	19.2	14.9	2.1
hasep1	3D long range	11/1	175.6	218.9	43.3	24.7	3.9
	2D-5pt	4/1	45.5	55.7	10.2	22.4	2.5
	3D-7pt	4/1	53.5	61.3	7.8	14.6	1.9
	triad	4/1	40.5	57.5	17.0	42.0	4.3
	scale	2/1	23.5	32.7	9.2	39.1	4.6
	add	3/1	32.0	45.0	13.1	40.6	4.4
	scalar product	2/0	17.0	26.4	9.5	55.3	4.7
	3D-27pt	4/1	114.1	92.5	-21.6	-18.9	-5.4
	vector sum	1/0	8.5	16.4	7.9	92.9	7.9
	DAXPY	2/1	24.5	33.3	8.9	35.9	4.4
	1D-3pt	2/1	26.5	47.4	20.9	78.9	10.5
	uxx stencil	9/1	121.5	151.3	29.8	24.5	3.3

Table 4.2.: Memory access penalty calculation per microarchitecture from: (bench. − prediction)/no. of read streams

[a]deviation = (benchmark − prediction)/prediction

4.1. Streaming Kernels

The following streaming kernels are typical examples used to benchmark the performance of memory systems. Their arithmetic intensity is small and they depend highly on the memory bandwidth and do not profit from caching, unless the data set is small enough to completely fit into a cache level. The data is therefore continuously streamed from memory to the CPU and back, hence the name.

DAXPY

```
double a[N], b[N];
double s;

for(int i=0; i<N; ++i)
    a[i] = a[i] + s * b[i];
```

The DAXPY kernel accumulates a scaled vector to another double precision vector, creating a linear combination of two vectors. It is often used in numerical codes and is therefore part of linear algebra libraries like BLAS [31].

ADD

```
double a[N], b[N], c[N];

for(int i=0; i<N; ++i)
    a[i] = b[i] + c[i];
```

This kernel is part of the STREAM benchmark collection [32], a standard memory bandwidth benchmark. It adds two double precision vectors and stores the result in a third. The write miss associated with the access to `a` will lead to an additional read stream due to write-allocate.

Scalar Product

```
double a[N], b[N];
double s;

for(int i=0; i<N; ++i)
    s += a[i] * b[i];
```

The scalar product is a common numerical operation (e.g., used to calculate the angle between two vectors), where the element-wise product of two vectors are summed up. It is also known as dot product or inner product. The (intermediate) result `s` is always kept in registers, so no write stream exists. It is also part of the BLAS library under the name of “DDOT”.

Scale

```
double a[N], b[N];
double s;

for(int i=0; i<N; ++i)
    a[i] = s * b[i];
```

This kernel scales one vector by a scalar floating point number and stores the result into another vector. It is also part of the STREAM benchmark collection. There are two read streams (one from `b` and one from write-allocate of `a`) and one write stream. `s` is expected to be kept in a register throughout all iterations.

Triad

```
double a[N], b[N], c[N], d[N];
double s;

for(int i=0; i<N; ++i)
    a[i] = b[i] + c[i] * d[i];
```

The Schönauer Vector Triad kernel is similar to the STREAM benchmark Triad, but with an additional read stream for `d` instead of a constant. It has the most unbalanced streaming kernel used in this evaluation, with four read streams (one due to write-allocate) and only one write stream.

Vector Sum

```
double a[N];
double s;

for(int i=0; i<N; ++i)
    s += a[i];
```

Vector sum is a simple reduction kernel, adding up all components of a vector to a single scalar. As with the scalar product reduction, there is no write stream going back to memory, because the result is kept in a register.

4.2. Stencil Codes

Stencil codes show peculiar access pattern into arrays, which arises from their relative off sets in multiple dimensions (see Chapter 1). They can benefit highly from caching, even if the complete dataset does not fit into any cache level. They are of special intrest, because many numerical solvers are based on these types of data accesses.

1D-3pt

```
double a[N], b[N], c;

for(int i=1; i<N-1; ++i)
    b[i] = c * (a[i-1] - 2.0*a[i] + a[i+1]);
```

This kernel is special in the sense that it fulfills the definition of both stencil and streaming kernels. Due to caching, only a single read stream will hit anything beyond the first cache level and in the results we will see the typical streaming pattern.

To get results from all cache levels, predictions and measurements are done in the range of `N` from 10^1 to 10^8 elements. This gives a total data size of 160 Byte to 1.5 GB (`a` and `b` combined) in memory.

2D-5pt

```
double a[M][N];
double b[M][N];
double s;

for(int j=1; j<M-1; ++j)
    for(int i=1; i<N-1; ++i)
        b[j][i] = ( a[j][i-1] + a[j][i+1]
                  + a[j-1][i] + a[j+1][i]) * s;
```

Because of the row major storage format, only the constant `N` influences the caching behavior, as can be seen in the layer-condition:

$$3 \cdot N \cdot 8\,\text{Byte} < \frac{C_k}{2}$$

with cache size C_k.

`M` is scaled to use at least a total of 1 GB of data and 16 rows. It is important that loops have a minimum number of iterations, otherwise the prediction assumptions would not be correct. The size of at least 1 GB is also important to fulfill the assumption that all data is residing in main memory. Hence, we choose:

$$\mathtt{M} = \max\left(\frac{1024^3\,\text{B}}{8\,\text{B} \cdot \mathtt{N}}, 16\right)$$

For this kernel we let N range from 10^1 to 10^7, in order to to get a complete overview of different cache and memory usages.

3D-7pt

```
double a[M][N][N];
double b[M][N][N];
double s;

for(int k=1; k<M-1; ++k)
    for(int j=1; j<N-1; ++j)
        for(int i=1; i<N-1; ++i)
            b[k][j][i] = ( a[k][j][i-1] + a[k][j][i+1]
                         + a[k][j-1][i] + a[k][j+1][i]
                         + a[k-1][j][i] + a[k+1][j][i]) * s;
```

We use the same basic principle as in the 2D-5pt case to calculate `M`, but needed to adapt the rule for the 3D case:

$$\mathtt{M} = \max\left(\frac{1024^3\,\mathrm{B}}{8\,\mathrm{B}\cdot\mathtt{N}^2}, 16\right)$$

For all following 3D kernels we used the same calculation of `M` based on `N` and made predictions and measurements from $N = 10^1$ to 10^4, in order to stay within the main memory limit while making use of all cache levels.

This adaptation is also reflected in the layer-condition:

$$3 \cdot N^2 \cdot 8\,\mathrm{Byte} < \frac{C_k}{2}$$

In addition to the 3D layer-condition, this also yields a 2D layer-condition or simply row-condition (explained in Section 4.3.2):

$$3 \cdot N \cdot 8\,\mathrm{Byte} < \frac{C_k}{2}$$

3D-27pt

```
double a[M][N][N];
double b[M][N][N];
double s;

for(int k=1; k<M-1; ++k)
    for(int j=1; j<N-1; ++j)
        for(int i=1; i<N-1; ++i)
            b[k][j][i] = ( a[k][j][i]
                         + a[k][j][i-1] + a[k][j][i+1]
                         + a[k][j-1][i] + a[k][j+1][i]
                         + a[k-1][j][i] + a[k+1][j][i]
                         + a[k][j-1][i-1] + a[k][j-1][i+1]
                         + a[k][j+1][i-1] + a[k][j+1][i+1]
                         + a[k-1][j][i-1] + a[k-1][j][i+1]
                         + a[k+1][j][i-1] + a[k+1][j][i+1]
                         + a[k-1][j-1][i] + a[k-1][j+1][i]
                         + a[k+1][j-1][i] + a[k+1][j+1][i]
                         + a[k-1][j-1][i-1] + a[k-1][j-1][i+1]
                         + a[k-1][j+1][i-1] + a[k-1][j+1][i+1]
                         + a[k+1][j-1][i-1] + a[k+1][j-1][i+1]
                         + a[k-1][j+1][i-1] + a[k-1][j+1][i+1]
                         ) * s;
```

This kernel is a higher demanding version of the 3D-7pt kernel, since there are nine instead of five streams coming from the L1 cache. The 3D layer-condition is the same as in the 3D-7pt case, but the 2D layer-condition changes a little:

$$3 \cdot 3 \cdot N \cdot 8\,\mathrm{Byte} < \frac{C_k}{2}$$

3D Long Range

```
double U[M][N][N];
double V[M][N][N];
double ROC[M][N][N];
double c0, c1, c2, c3, c4, lap;

for(int k=4; k < M-4; k++) {
    for(int j=4; j < N-4; j++) {
        for(int i=4; i < N-4; i++) {
            lap = c0 * V[k][j][i]
                + c1 * ( V[ k ][ j ][i+1] + V[ k ][ j ][i-1])
                + c1 * ( V[ k ][j+1][ i ] + V[ k ][j-1][ i ])
                + c1 * ( V[k+1][ j ][ i ] + V[k-1][ j ][ i ])
                + c2 * ( V[ k ][ j ][i+2] + V[ k ][ j ][i-2])
                + c2 * ( V[ k ][j+2][ i ] + V[ k ][j-2][ i ])
                + c2 * ( V[k+2][ j ][ i ] + V[k-2][ j ][ i ])
                + c3 * ( V[ k ][ j ][i+3] + V[ k ][ j ][i-3])
                + c3 * ( V[ k ][j+3][ i ] + V[ k ][j-3][ i ])
                + c3 * ( V[k+3][ j ][ i ] + V[k-3][ j ][ i ])
                + c4 * ( V[ k ][ j ][i+4] + V[ k ][ j ][i-4])
                + c4 * ( V[ k ][j+4][ i ] + V[ k ][j-4][ i ])
                + c4 * ( V[k+4][ j ][ i ] + V[k-4][ j ][ i ]);
            U[k][j][i] = 2.f * V[k][j][i] - U[k][j][i]
                       + ROC[k][j][i] * lap;
}}}
```

The 3D long range stencil has a radius of four in each of the six coordinate directions. The far off-center accesses in `k` direction lead to an extremely high demand for cache size, since the 3D layer-condition can only be satisfied, if $9 \cdot \mathtt{N}^2$ elements fit into the cache. A depiction of the different layer-conditions that can be fulfilled by this kernel is shown in Figure 4.2. The 2D layer-condition is described by

$$9 \cdot N \cdot 8\,\text{Byte} < \frac{C_k}{2}$$

This stencil was originally used by Malas et al. [33] at KAUST and for performance analysis by Stengel et al. [4].

UXX

```
double u1[M][N][N];
double d1[M][N][N];
double xx[M][N][N];
double xy[M][N][N];
double xz[M][N][N];
double c1, c2, d, dth;

for(int k=2; k<M-2; k++) {
    for(int j=2; j<N-2; j++) {
        for(int i=2; i<N-2; i++) {
            d = 0.25*(d1[ k ][j][i] + d1[ k ][j-1][i]
                    + d1[k-1][j][i] + d1[k-1][j-1][i]);
            u1[k][j][i] = u1[k][j][i] + (dth/d)
             * ( c1*(xx[ k ][ j ][ i ] - xx[ k ][ j ][i-1])
               + c2*(xx[ k ][ j ][i+1] - xx[ k ][ j ][i-2])
               + c1*(xy[ k ][j+1][ i ] - xy[ k ][j-1][ i ])
               + c2*(xy[ k ][j+1][ i ] - xy[ k ][j-2][ i ])
               + c1*(xz[ k ][ j ][ i ] - xz[k-1][ j ][ i ])
               + c2*(xz[k+1][ j ][ i ] - xz[k-2][ j ][ i ]));
}}}
```

This stencil is part of a simulation code for dynamic rupture and earthquake wave propagation [34]. It is special because of the unsymmetric access pattern in all three dimensions, and because of the “expensive” divide operation in the loop body. See [4] for a detailed analysis. The 3D layer-condition is described

by the following inequation:

$$4 \cdot N^2 \cdot 8\,\text{Byte} < \frac{C_k}{2}$$

and the 2D layer-condition consequently by:

$$4 \cdot N \cdot 8\,\text{Byte} < \frac{C_k}{2}$$

4.3. Results

In Figures 4.3 and 4.4, we present the results of kerncraft predictions and measurements side-by-side on three different microarchitectures. “phinally” refers to the system described in Table 2.1 as “Sandy Bridge”, “ivyep1” is the “Ivy Bridge” system and “hasep1” is the described “Haswell” system. All shown kernels have been described and their underlying code presented in Sections 4.1 and 4.2. The vertical axis in the plots represents cycles per cache-line, the usual metric for ECM predictions. The horizontal axis represents number of elements on the inner dimension `N`. Thus “higher” numbers (on the vertical axis) are slower, since more cycles need to be executed per cache-line of work. For better comparison, the scale of the axes is the same on each row.

The background coloring represents the stacked contributions from the ECM prediction, from blue (bottom) to orange (top): T_{nOL}, T_{L1L2}, T_{L2L3} and T_{L3MEM}. The red line is the T_{OL} contribution from the in-core analysis.

Measurements are represented by black crosses, which are evenly distributed throughout the plot range. Some datapoints towards the beginning are missing, because the measured results lie above the plot’s upper limit. This was done intentionally in order to focus on the relevant data range, since L1 measurements are often dominated by other effects, mostly due to short loop lengths.

4.3.1. Streaming Kernels

The plots for streaming kernels in Figure 4.3 (and for the 1D-3pt stencil from Figure 4.4) show the behavior of the kernels if the complete data set can fit into different memory hierarchy levels. The ECM model in kerncraft is based on the assumption that the dataset does not completely fit into any cache level and therefore resides in memory for the most part. Since this assumption is not necessarily true for streaming kernels, we interpret the results slightly differently than with stencil codes. One prediction already contains all results needed to anticipate the performance at all levels, by adding up the relevant contributions and ignoring the rest of the memory hierarchy. For this reason a perfect result would be that each step of the measurement results (black crosses) corresponds to a prediction level (colored background), which can be seen easily in the graphical representation. Each step is located at the maximum size of a cache level, although not always a perfectly straight edge can be seen due details in cache associativities and replacement strategies.

Measurements with very few elements in the L1 cache yield bad results, since loop overheads are more dominant. Especially interesting is the straight “line” of measurement points present in all plots between $N = 10 \ldots 30$. This is due to loop unrolling and vectorization, which leads to significant overhead if the input data is not a multiple of the CPU’s supported vector length and the compiler’s chosen loop unrolling length. In these cases, a “fallback” or “remainder” loop needs to be used, which is usually much slower because of missing vectorization and unrolling. With increasing number of elements, the remainder loop needs to be iterated more often and performance decreases, until the number is a multiple again where performance is at a “sweet spot”.

In Figure 4.3, we can see that the accuracy of T_{nOL} and T_{OL} is very good. L2 (T_{L1L2}) and L3 (T_{L2L3}) predictions are also quite accurate, but vary with the underlying kernel code. For example: ADD and TRIAD on phinally and ivyep1 are almost perfect matches of prediction and measurement from $N = 10^2$ to 10^6. hasep1 does not match as well, but is still within a couple of cycles of the prediction. The good prediction of T_{L3MEM} on ivyep1 and hasep1 comes from the correcting penalty cycles, which are tailored

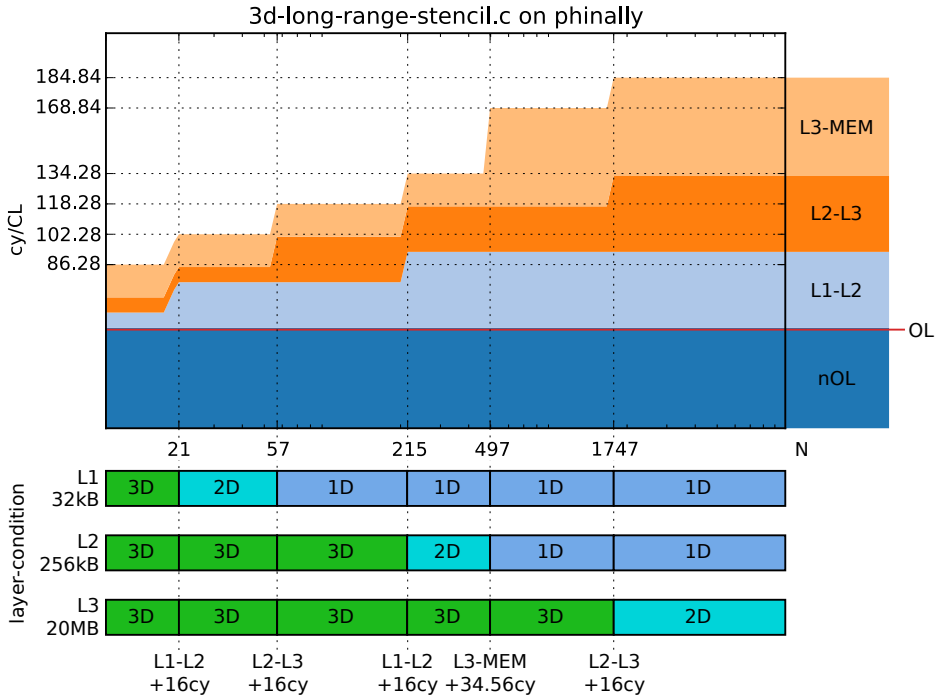


Figure 4.1.: Detailed plot of the ECM prediction for the 3D Long Range Stencil on phinally, with comparison to fulfilled layer-conditions on each cache level.

to the results we have at hand, as was explained in the beginning of this chapter. The memory prediction on phinally, was not corrected and is purely based on measurements with stream-like benchmarks. In general, we can say that for codes with balanced write and load streams, the prediction is quite accurate, but pure load based codes (vector sum and scalar product) seem to be influenced by effects that we do not model.

4.3.2. Stencil Codes

The predictions of stencil codes in Figure 4.4 show steps, which reflect the different layer-conditions being violated in different caches in turn. It is assumed that the complete dataset is held in the main memory and caches can only hold a subset of that. If we ignore the 1D-3pt stencil, which behaves more like a streaming kernel, we can see that the ECM predications are quite accurate. The ideal measurement/prediction result would be that all black crosses (measurement results) are residing right on the top edge of the colored background (prediction). This can be written as $T_{\text{expected measurement}} = T_{\text{nOL}} + T_{\text{L1L2}} + T_{\text{L2L3}} + T_{\text{L3MEM}}$. On the far left side, we see the prediction for the smallest data set ($N = 10$), which corresponds to the layer-condition met on the first level cache (L1). The measurements are nowhere near the predictions in all cases, because such short loops generate a large overhead in relation to the actual work. When looking further to the right, increasing N, we see steps in the predictions when layer-conditions are not met anymore. The L1 layer-condition is not fulfilled by increasing N just slightly, for example at $N = 10^3$ with the 2D-5pt stencil. On the far right, no layer-condition is fulfilled anymore and most data needs to come from memory. With each step between the far left and far right, another layer-condition is not met anymore. On the more complex kernels, like UXX, there are more steps than cache levels, because the three dimensional nature of the stencil leads to three layer-conditions which can be fulfilled in each cache level. The far left and far right still correspond to L1 layer-condition and no layer-condition, but due to the cache simulation, intermediate steps are also predicted and lead to a more fine-grained prediction.

Figure 4.1 depicts the origin of the steps in the ECM prediction using the example of the 3D Long Range

Stencil on phinally. Bellow the prediction plot are the conditions that are fulfilled on each cache level. “3D” means that the full layer-condition in all three dimensions is fulfilled and requires at least $9 \cdot N^2 \cdot 8$ Byte of cache size for `V`. The 2D layer-condition only requires the elements lying in one plane to be stored and like with 2D stencils, this condition calls for $9 \cdot N \cdot 8$ Byte cache size. The one dimensional condition is always fulfilled, since nine consecutive elements can always be stored in the cache. Figure 4.2 illustrates the three different layer-conditions, which can be found in any three dimensional stencil.

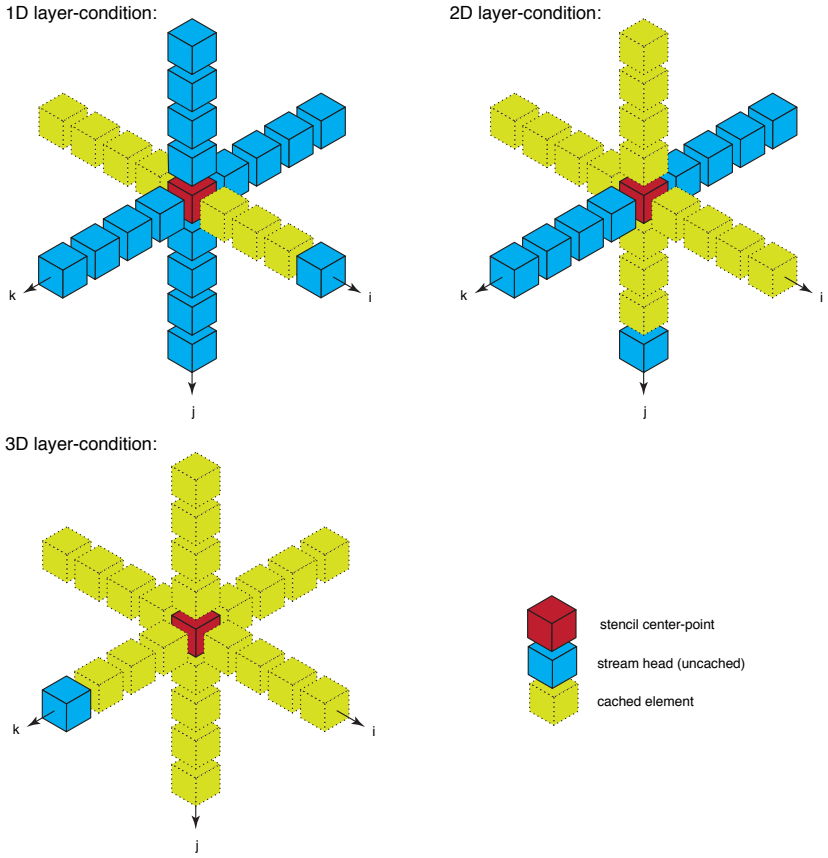


Figure 4.2.: 3D layer-conditions inherent to all three dimensional stencil codes. *i* refers to the inner loop, and *k* to the outer loop. The depicted stencil is the 3D long-range-stencil from Section 4.2.

Coming back to the 1D-3pt stencil, we can see that the predictions do not match the measurements very well. Even the IACA prediction does not match up with the measurement in L1, which is striking because this is very accurate for all other streaming kernels. If we add the offset between IACA prediction and measurement in L1 (roughly 5 cycles) to all other predictions we get the following: L2 should be around 17 cy/CL and we measure 19, which is not perfect but acceptable. L3 would be at 23 cy/CL, but we get 28 cy/CL and the memory measurement is also off by another 5 cycles per cache-line. We were unfortunately unable to understand the discrepancies, which involve all tested architectures. We have tried to see if it only happens in load bound versions of the code, but even with SSE2 instructions measurements and predictions do not match up.

Overall, predictions and measurements in Figure 4.4 match up quite well on the Sandy Bridge architecture. On Ivy Bridge and Haswell, penalty cycles were used to correct for discrepancies of the T_{L3MEM} latency. In some cases this lead to an over correction, e.g., on 3D-27pt and 3D-7pt on ivyep1 and hasep1, as well as 2D-5pt on hasep1.

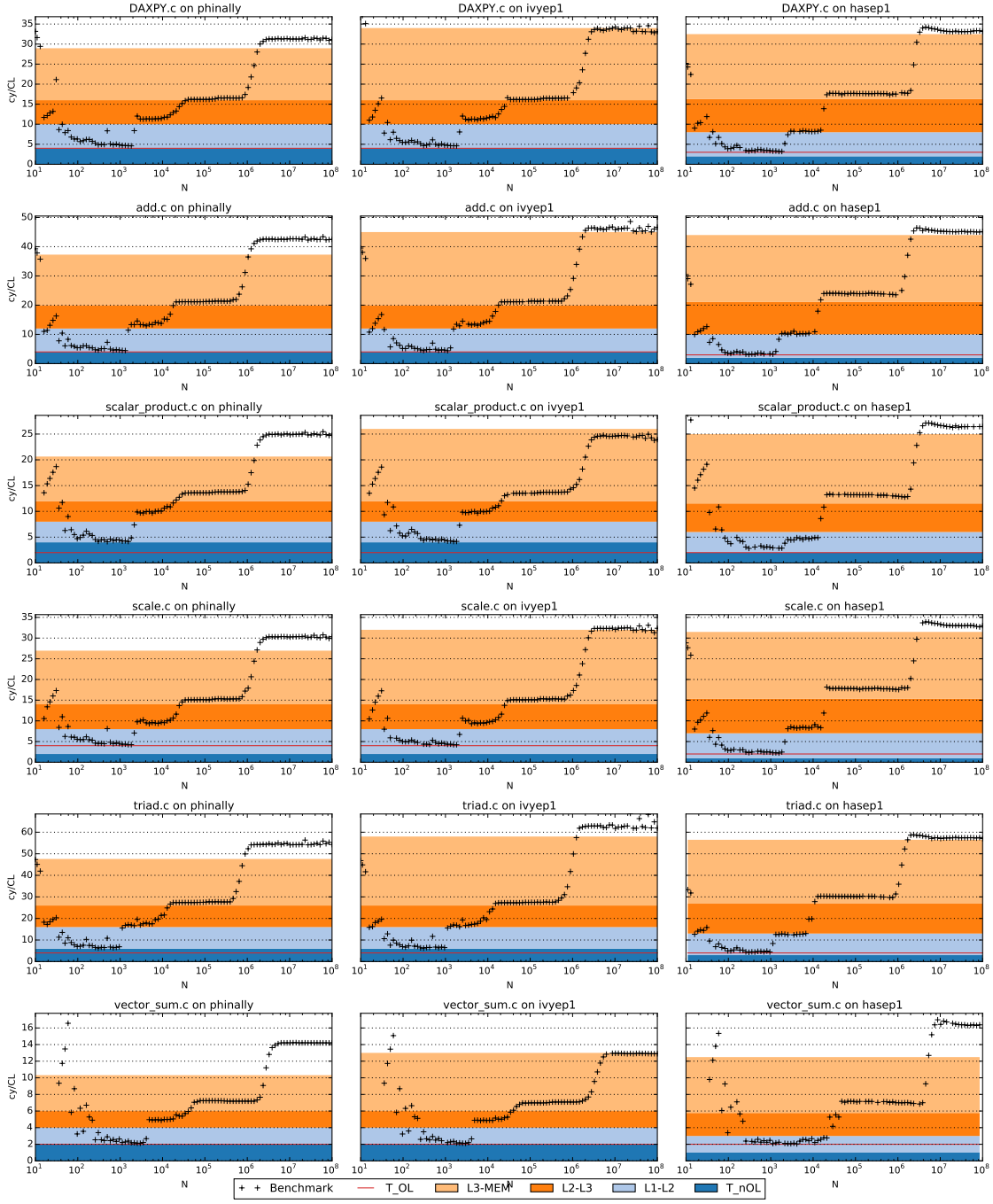


Figure 4.3.: Comparison of ECM model predictions by kerncraft and measurements for the streaming kernels (Section 4.1) across all levels of the memory hierarchy. The different color bands visualize contributions to the prediction from L1 (at bottom), L2 and L3 cache levels and main memory (at top). The red line is the overlapping time T_{OL}. Measurements are marked by black plus-signs.

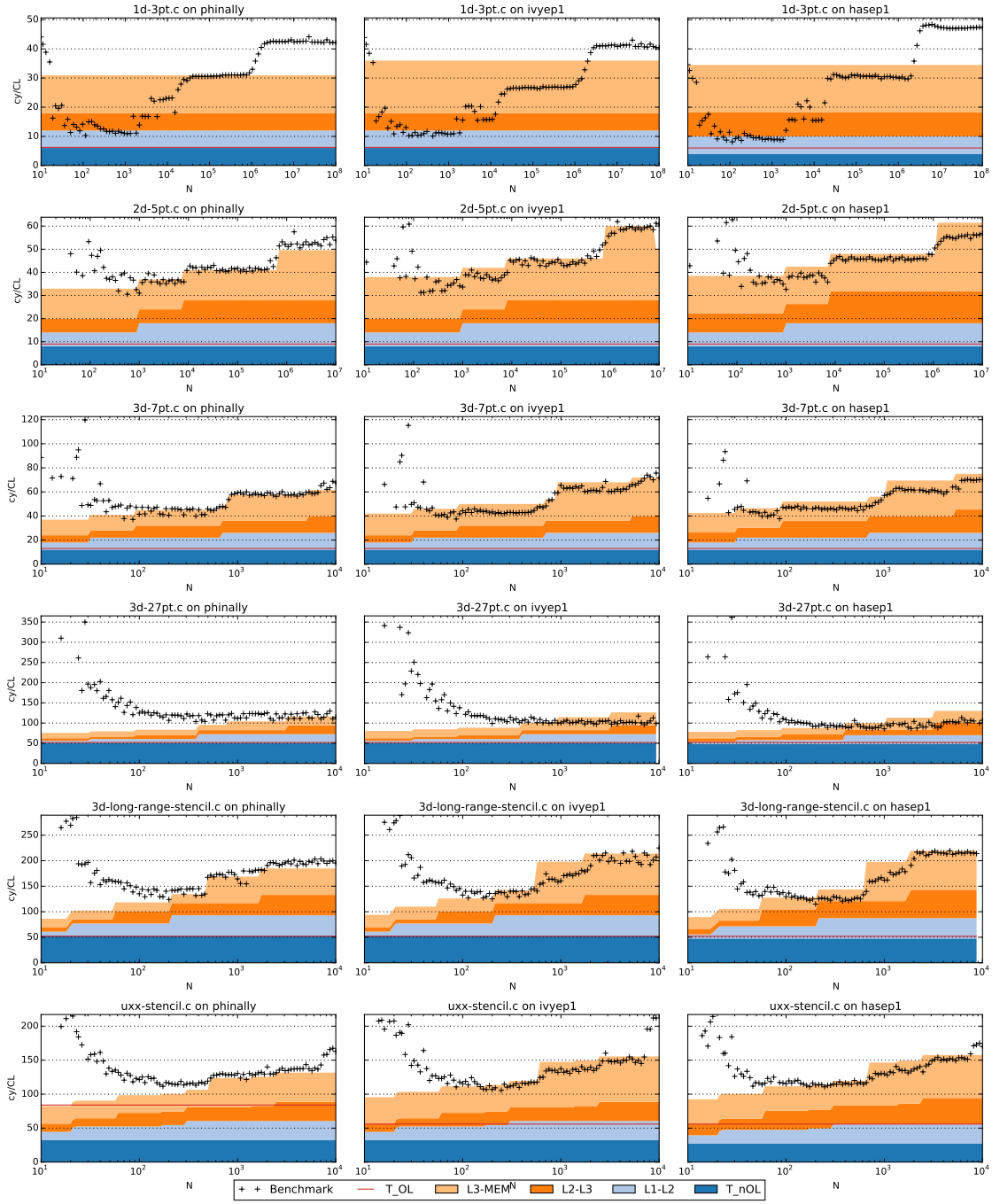


Figure 4.4.: Comparison of ECM model predictions by kerncraft and measurements for the stencil kernels (Section 4.2) with all data originating from main memory, but layer-conditions fulfilled at different cache levels. The different color bands visualize contributions to the prediction from L1 (at bottom), L2 and L3 cache levels and main memory (at top). The red line is the overlapping time T_{OL}. Measurements are marked by black plus-signs.

5 Conclusion and Future Work

We have shown that an automatic analysis of streaming and stencil loop kernels using analytical models is possible and allows a detailed view on performance in relation to a given hardware architecture. Our "kerncraft" tools will be especially helpful to simulation developers, performance engineers and computer scientists interested in understanding performance behavior of loop kernel based codes. We provide a toolkit that will accurately predict the single-core performance of loop kernels, relying for the most part on static analysis and documented hardware specifications, and requires no execution and measurements beyond the inputs for the underlying Execution-Cache-Memory (ECM) and Roofline models. This allows quick and easy evaluation of prototype code on a variety of hardware architectures and a detailed analysis of performance gains to be expected of algorithmic and hardware changes.

Although the kerncraft tool is able to construct both the ECM and Roofline model, our analysis focused on the ECM model and its accuracy when applied to different streaming and stencil kernels. In load dominated test-cases predictions were not perfectly accurate, due to under estimation of last level cache to main memory transfers, since the ECM model currently lacks a precise model for memory throughput. It is therefore based on measurements rather than well-founded architecture analysis. For this reason, penalty cycles have been introduced and yield acceptable results. This area is currently under investigation, especially since the discrepancies (and penalty cycles) grew larger with more recent microarchitecture generations. Preliminary results have been published [23].

Development on kerncraft will continue along with the advancement of the ECM model. Especially the interoperability, usability and presentation of results are key challenges for future work. Currently, we only support Intel processors with Nehalem microarchitecture or newer, due to dependency on the Intel Automatic Code Analyzer (IACA) and the Intel Compiler (ICC). To lift that restriction, an in-core execution simulation needs to be developed as a replacement for IACA. The dependency on ICC will be changed and replaced by a generic compiler interface allowing the use of any available compiler. Support for many-core based architectures, like GPUs or Intel Xeon Phis, is tricky, since it depends foremost on the adaptation of the ECM model to such architectures. The Roofline part of kerncraft will be less problematic.

Some work will be done on better presentation of results, since this will make the output more complete and easier to interpret. One discussed strategy is to always compile a full report on all data set sizes

of interest (all different layer-condition fulfilled), as well as a multi-core scaling analysis. A more detailed plotting scheme for data set size sweeps is planned, where each “step” in the plot comes with a detailed explanation of the cache hits and misses on each level. The report could also give blocking suggestions to the programmer, including suggested block sizes and their predicted speed ups.

The cache simulator is currently very rudimentary and could be extended to simulate aliasing effects due to cache associativity, and different cache replacement policies.

Another approach to gather the cache reuse information is the interception of memory accesses while running the code. There are several papers on this technique, which can be an addition to the kernel code static analysis, to validate results and predict behavior of more complex codes embedded within large projects.

To make the toolkit more widely accessable, we plan to offer a hosted version, where no local installation is necessary and the automatic code analysis is offered as a service for interested parties via an online web service.

Bibliography

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

[2] Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Parallel Processing and Applied Mathematics*, pages 615–624. Springer Science + Business Media, 2010. doi: 10.1007/978-3-642-14390-8_64.

[3] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 2014. doi: 10.1002/cpe.3180.

[4] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751240.

[5] Bill Gropp. Engineering for performance in high performance computing. Presentation at Platform for Advanced Scientific Computing (PASC14), June 2014. URL `https://www.youtube.com/watch?v=sadfSARXSC0`.

[6] Chen Ding and Yutao Zhong. Reuse distance analysis. Technical report, University of Rochester, New York, 2001. URL `http://www.cs.rochester.edu/u/cding/Documents/Publications/TR741.pdf`.

[7] Xu Liu and John Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 183–193. IEEE, 2013. doi: 10.1109/ispass.2013.6557169.

[8] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *Lecture Notes in Computer Science*, pages 129–148. Springer Science + Business Media, 2015. doi: 10.1007/978-3-319-17248-4_7.

[9] Sri Hari Krishna Narayanan, Boyana Norris, and Paul D. Hovland. Generating performance bounds from source code. In *39th International Conference on Parallel Processing Workshops*, pages 197–206. Institute of Electrical & Electronics Engineers (IEEE), 2010. doi: 10.1109/icppw.2010.37.

[10] John von Neumann. *First Draft of a Report on the EDVAC*. 1945. doi: 10.1007/978-3-642-61812-3_30.

[11] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2014. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[12] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21"(9): 948–960, 1972. doi: 10.1109/TC.1972.5009071.

[13] *Locality optimizations on ccNUMA architectures*, pages 185–202. CRC Press, 2015/06/29 2010. ISBN 978-1-4398-1192-4. doi: 10.1201/EBK1439811924-c8.

[14] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[15] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 65–74. IEEE, 2013. doi: 10.1109/rtas.2013.6531080.

[16] Johannes Hofmann. Extending the ECM Model. Presentation at Seminar on Multi-Core Simulation, April 2015. URL http://faui36a.informatik.uni-erlangen.de/hofmann/permanent/ecm_haswell.pdf.

[17] Israel Hirsh. Intel architecture code analyzer. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

[18] likwid – lightweight performance tools. URL http://tiny.cc/LIKWID.

[19] Willi Schönauer. *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*. Self-edition, 2000. URL http://www.rz.uni-karlsruhe.de/~rx03/book.

[20] H. T. Kung. Memory requirements for balanced computer architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 49–54, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0719-X. doi: 10.1145/17356.17362. URL http://dl.acm.org/citation.cfm?id=17407.17362.

[21] R. W. Hockney and I. J. Curington. $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10(3):277–286, 1989. doi: 10.1016/0167-8191(89)90100-2.

[22] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334 – 358, 1988. ISSN 0743-7315. doi: 10.1016/0743-7315(88)90002-0.

[23] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Performance analysis of the kahan-enhanced scalar product on current multicore processors. *Accepted for PPAM 2015, the 11th International Conference on Parallel Processing and Applied Mathematics*, abs/1505.02586, September 3-6, 2015, Krakow, Poland. URL http://arxiv.org/abs/1505.02586.

[24] Georg Hager. Insight into stencil performance by analytic modeling. Presentation at Seminar on Advanced Stencil Code Engineering, April 2015. URL http://blogs.fau.de/hager/files/2010/09/Dagstuhl_Stencils_Hager_2015.pdf.

[25] Free Software Foundation. GNU Affero General Public License. URL http://www.gnu.org/licenses/agpl-3.0.html.

[26] PyPI - the Python Package Index. URL https://pypi.python.org/pypi.

[27] ISO. ISO C Standard 1999. Technical report, 1999. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. ISO/IEC 9899:1999 draft.

[28] Gerhard Wellein. Programming techniques for supercomputers. Lecture, 2014. URL http://moodle.rrze.uni-erlangen.de/course/view.php?id=306.

[29] Eli Bendersky. pycparser. URL https://github.com/eliben/pycparser.

[30] YAML Ain't Markup Language. URL http://yaml.org.

[31] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979. doi: 10.1145/355841.355847.

[32] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[33] Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, 10 2015. doi: 10.1137/140991133. URL http://arxiv.org/abs/1410.3060.

[34] Matthias Christen and Olaf Schenk. A performance study of an anelastic wave propagation code using auto-tuned stencil computations. *Procedia Computer Science*, 9:956–965, 2012. doi: 10.1016/j.procs.2012.04.102.

A

FROM IACA TO T_{OL} AND T_{NOL}

In order to explain the interpretation of IACA results, we will use the 2D-5pt Jacobi kernel (with code seen in Listing 1.2) on the Sandy Bridge architecture (as described in Table 2.1). This method uses the same procedure used in kerncraft to derive T_{OL} and T_{nOL} for the ECM and RooflineIACA models to derive the CPU bottleneck performance.

As explained in Section 3.2.3, before the analysis can take place, the code needs to be compiled using the appropriate flags, found in the machine description (see Appendix B), in order to produce an optimized assembly code. Also, the relevant loop block needs to be marked using in the assembly code, e.g., with the help of `iaca_marker.py` described in Appendix C. The assembly is then compiled to an non-working—due to the marker sequences—executable.

With an appropriately prepared executable, we can run IACA and will receive the output seen in Listing A.1. From here on, we need to figure out three things: the loop unroll size (how many iterations of the high-level code are folded into one iteration in the assembly), total throughput cycles attributed to LOAD instructions and total throughput cycles attributed to non-load instructions.

```
Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - kernels/2d-5pt.iaca_marked
Binary Format - 64Bit
Architecture  - SNB
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 18.90 Cycles       Throughput Bottleneck: FrontEnd, PORT2_AGU, PORT3_AGU

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0  -  DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |
---------------------------------------------------------------------------
| Cycles | 10.1   0.0 | 12.0 | 18.0   16.0 | 18.0   16.0 | 8.0 | 11.9 |
---------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |            Ports pressure in cycles             |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
```

```
30 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm2, xmmword ptr [rbx+rdi*8]
31 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm3, xmmword ptr [rbx+rdi*8+0x10]
32 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm14, xmmword ptr [rbx+rdi*8+0x20]
33 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm15, xmmword ptr [rbx+rdi*8+0x30]
34 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm6, xmmword ptr [r15+rdi*8+0x8]
35 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm9, xmmword ptr [r14+rdi*8+0x8]
36 |   2    | 0.2      |     | 1.0   1.0 |           |     | 0.9 | CP | vinsertf128 ymm4, ymm2, xmmword ptr [rbx+rdi*8+0x10], 0x1
37 |   2    | 0.2      |     |           | 1.0   1.0 |     | 0.8 | CP | vinsertf128 ymm5, ymm3, xmmword ptr [rbx+rdi*8+0x20], 0x1
38 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm7, ymm4, ymm5
39 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm4, xmmword ptr [r15+rdi*8+0x28]
40 |   2    | 0.2      |     |           | 1.0   1.0 |     | 0.8 | CP | vinsertf128 ymm2, ymm14, xmmword ptr [rbx+rdi*8+0x30], 0x1
41 |   2    | 0.6      |     | 1.0   1.0 |           |     | 0.3 | CP | vinsertf128 ymm3, ymm15, xmmword ptr [rbx+rdi*8+0x40], 0x1
42 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm5, ymm2, ymm3
43 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm3, xmmword ptr [r15+rdi*8+0x48]
44 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm15, xmmword ptr [r15+rdi*8+0x68]
45 |   2    | 0.3      |     |           | 1.0   1.0 |     | 0.6 | CP | vinsertf128 ymm8, ymm6, xmmword ptr [r15+rdi*8+0x18], 0x1
46 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm10, ymm7, ymm8
47 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm7, xmmword ptr [r14+rdi*8+0x28]
48 |   2    | 0.4      |     |           | 1.0   1.0 |     | 0.6 | CP | vinsertf128 ymm6, ymm4, xmmword ptr [r15+rdi*8+0x38], 0x1
49 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm8, ymm5, ymm6
50 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm6, xmmword ptr [r14+rdi*8+0x48]
51 |   2    | 0.8      |     |           | 1.0   1.0 |     | 0.2 | CP | vinsertf128 ymm11, ymm9, xmmword ptr [r14+rdi*8+0x18], 0x1
52 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm12, ymm10, ymm11
53 |   1    | 1.0      |     |           |           |     |     |    | vmulpd ymm13, ymm0, ymm12
54 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm12, xmmword ptr [rbx+rdi*8+0x40]
55 |   2    |          |     |           | 1.0       | 2.0 |     | CP | vmovupd ymmword ptr [r9+rdi*8+0x8], ymm13
56 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm13, xmmword ptr [rbx+rdi*8+0x50]
57 |   2    | 0.2      |     | 1.0   1.0 |           |     | 0.8 | CP | vinsertf128 ymm9, ymm7, xmmword ptr [r14+rdi*8+0x38], 0x1
58 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm10, ymm8, ymm9
59 |   1    | 1.0      |     |           |           |     |     |    | vmulpd ymm11, ymm0, ymm10
60 |   2    |          |     | 1.0       |           | 2.0 |     | CP | vmovupd ymmword ptr [r9+rdi*8+0x28], ymm11
61 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm11, xmmword ptr [rbx+rdi*8+0x60]
62 |   2    | 0.1      |     | 1.0   1.0 |           |     | 0.9 | CP | vinsertf128 ymm14, ymm12, xmmword ptr [rbx+rdi*8+0x50], 0x1
63 |   1    |          |     |           | 1.0   1.0 |     |     | CP | vmovupd xmm12, xmmword ptr [rbx+rdi*8+0x70]
64 |   2    | 0.2      |     | 1.0   1.0 |           |     | 0.8 | CP | vinsertf128 ymm2, ymm13, xmmword ptr [rbx+rdi*8+0x60], 0x1
65 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm4, ymm14, ymm2
66 |   2    | 0.4      |     |           | 1.0   1.0 |     | 0.6 | CP | vinsertf128 ymm13, ymm11, xmmword ptr [rbx+rdi*8+0x70], 0x1
67 |   2    | 0.2      |     | 1.0   1.0 |           |     | 0.8 | CP | vinsertf128 ymm14, ymm12, xmmword ptr [rbx+rdi*8+0x80], 0x1
68 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm2, ymm13, ymm14
69 |   2    | 0.6      |     |           | 1.0   1.0 |     | 0.4 | CP | vinsertf128 ymm5, ymm3, xmmword ptr [r15+rdi*8+0x58], 0x1
70 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm7, ymm4, ymm5
71 |   0*   |          |     |           |           |     |     |    | nop
72 |   1    |          |     | 1.0   1.0 |           |     |     | CP | vmovupd xmm4, xmmword ptr [r14+rdi*8+0x68]
73 |   2    | 0.6      |     |           | 1.0   1.0 |     | 0.4 | CP | vinsertf128 ymm3, ymm15, xmmword ptr [r15+rdi*8+0x78], 0x1
74 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm5, ymm2, ymm3
75 |   2    | 0.5      |     | 1.0   1.0 |           |     | 0.5 | CP | vinsertf128 ymm8, ymm6, xmmword ptr [r14+rdi*8+0x58], 0x1
76 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm9, ymm7, ymm8
77 |   1    | 1.0      |     |           |           |     |     |    | vmulpd ymm10, ymm0, ymm9
78 |   2    |          |     |           | 1.0       | 2.0 |     | CP | vmovupd ymmword ptr [r9+rdi*8+0x48], ymm10
79 |   2    | 0.2      |     |           | 1.0   1.0 |     | 0.8 | CP | vinsertf128 ymm6, ymm4, xmmword ptr [r14+rdi*8+0x78], 0x1
80 |   1    |          | 1.0 |           |           |     |     |    | vaddpd ymm7, ymm5, ymm6
81 |   1    | 1.0      |     |           |           |     |     |    | vmulpd ymm8, ymm0, ymm7
82 |   2    |          |     | 1.0       |           | 2.0 |     | CP | vmovupd ymmword ptr [r9+rdi*8+0x68], ymm8
83 |   1    | 0.2      |     |           |           |     | 0.8 |    | add rdi, 0x10
84 |   1    |          |     |           |           |     | 1.0 |    | cmp rdi, r8
85 |   0F   |          |     |           |           |     |     |    | jb 0xffffffffffffffeaa
86 | Total Num Of Uops: 74
```

Listing A.1: IACA output for 2D-5pt Jacobi kernel

The loop unroll size is extracted by analyzing the end of the loop code, line 83 in the listing. Here, we see that the loop counter `rdi` is increased by 0*x*10 or 16. Thus, one iteration in the assembly code is equivalent to 16 iterations in the high-level code and handles the workload equivalent of two cache-lines. Therefore, we will need to divide all cycle counts by two, to normalize them to a "per cache-line" prediction.

To get the throughput cycle count, we will look at the port summary towards the top of the output (lines 11 to 16). From the Sandy Bridge architecture description in Section 2.1, we know that ports 1 and 2 handle LOAD instructions. In IACA they are split into a data pipe ("D") and non-data pipe ("2" and "3") part. We are interested only in the data pipe part for the non-overlapping contribution, therefore we have 16 cycles for two cache-lines and $T_{\text{nOL}} = 8$ cy per cache-line.

By taking the worst (maximum) value over all other ports (including "2" and "3"), we get 18 cycles for arithmetic and overlapping operations. Thus, we get $T_{\text{OL}} = 9$ cy per cache-line.

B
MACHINE DESCRIPTION IN KERNCRAFT

To pass the information about the hardware specifications and supported features `kerncraft`, a *machine description file* is required. To help the users with creating one, `likwid_auto_bench.py` will gather information using LIKWID tools. Since not everything can be extracted or benchmarked, it is necessary to provide additional information by hand from vendor documentation.

We will go through the steps of creating a machine description file for the Sandy Bridge system described in Table 2.1, also known as phinally.

B.1. File Format and Required Information

The machine description file is formatted according to the YAML [30] specifications. It contains description of the hardware as well as benchmark data generated with `likwid-bench`.

The basic description (excluding the benchmarks section) is presented in Listing B.1. In addition to the YAML specifications, it is possible to have a unit associated with the values (e.g., `2.7 GHz` on line 1), and this will automatically translate the value to the corresponding numerical value (e.g., 2700000000 Hz). Supported are the SI-prefixes kilo (`k`), Mega (`M`), Giga (`G`), Terra (`T`), Exa (`E`), Zetta (`Z`) and Yotta (`Y`).

```
1  clock: 2.7 GHz
2  cores per socket: 8
3  model type: Intel Core SandyBridge EP processor
4  model name: Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz
5  sockets: 2
6  threads per core: 2
7  cacheline size: 64 B
8  icc architecture flags: [-xAVX]
9  micro-architecture: SNB
10 FLOPs per cycle:
11     SP: {total: 8, ADD: 4, MUL: 4}
12     DP: {total: 4, ADD: 2, MUL: 2}
13 overlapping ports: ["0", "0DV", "1", "2", "3", "4", "5"]
```

```
14 non-overlapping ports: ["2D", "3D"]
15 memory hierarchy:
16 - {cores per group: 1, cycles per cacheline transfer: 2,
17   groups: 16, level: L1, bandwidth: null, size per group: 32.00
18     kB, threads per group: 2}
19 - {cores per group: 1, cycles per cacheline transfer: 2,
20   groups: 16, level: L2, bandwidth: null, size per group: 256.00
21     kB, threads per group: 2}
22 - {cores per group: 8, cycles per cacheline transfer: null,
23   groups: 2, level: L3, bandwidth: 40 GB/s, size per group: 20.00
24     MB, threads per group: 16}
25 - {cores per group: 8, cycles per cacheline transfer: null,
26   level: MEM, bandwidth: null, size per group: null, threads per group: 16}
```

Listing B.1: Machine description file (except for benchmark section) for Intel Sandy Bridge based CPU.

Most entries are self-explanatory and we will only go into detail about the more unintuitive ones:

- `clock` is the CPU baseclock during execution. This should not be the maximum clock available through turbo mode, because it can not be assumed to be always available and would not necessarily yield reproducible results.
- `model type` (line 3) and `model name` (line 3) are purely for textual output purposes.
- `icc architecture flags` (line 8) is a list of command line arguments passed to the ICC compiler, e.g., if only the analysis of SSE based optimizations are of interest, replace `-xAVX` with the `-xSSE`. Multiple arguments can be used when separated by comma.
- `micro-architecture` (line 9) is the abbreviation used by IACA to describe the underlying micro-architecture. Supported are `NHM` for Nehalem, `WSM` for Westmere, `SNB` for Sandy Bridge, `IVB` Ivy Bridge and `HSW` for Haswell micro-architectures.
- `FLOPs per cycle` (line 10-12) contains a dictionary of the number of FLOPs that can be executed per cycle (in a throughput sense). It is divided into single-precision (`SP`), double-precision (`DP`) and differentiates between addition (`ADD`), multiplication (`MUL`) and fused-multiply-add (`FMA`). The `total` is required, because sometimes only one or the other can be used at the same time (e.g., on the Haswell architecture). If `FMA` is not given, it is assumed unsupported.
- `overlapping ports` (line 13) are the port names, as returned by IACA, contributing to the T_{OL} in the ECM model, representing arithmetic and store operations. Some ports are subdivided in IACA and need separate mentioning (e.g., `0` and `0DV`, compare to raw IACA output line 13 in Listing A.1).
- `non-overlapping ports` (line 14) these are the remaining ports or subports, not listed under `overlapping ports`, responsible for LOAD instructions and contributing to T_{nOL}.
- `memory hierarchy` (line 15-26) contains the information about the sizes and theoretical bandwidths of the cache-levels and main memory. Each entry is one memory level, appearing in decreasing order from first level cache to main memory. The bandwidth information given at each level refers to the data paths between itself and the lower level, the main memory entry containing therefore no such information. Subentries within each memory hierarchy entries are the following:
  - `level` is an abbreviatory name used for textual output
  - `core per group` and `threads per group` are the number of physical cores and virtual threads sharing one physical cache
  - `groups` are the total number of physically distinct instances of the cache per socket.
  - `size per group` is the size of each physical instance of the cache. The memory size is not of interest and can be left at `null`.
  - `cycles per cacheline transfer` is the number of cycles it takes to transfer one cache-line in the throughput or streaming case (ignoring latencies). It can be understood as an inverse asymptotic bandwidth. In the last level cache, this entry can be used to impose an additional penalty per read stream between the last level cache and main memory, where otherwise only the memory bandwidth is considered.

In addition to the described white-paper information about the CPU, benchmarked information is considered for the memory interface to last level cache bandwidths. Descriptions of the benchmarked kernels and results for single- and multi-core, as well as different threads per core counts, are collected and stored in the benchmark section. This part is automatically generaded and it is not meant to be changed manually. The collected results are used for finding a closely matching read and write stream pattern and better predict the memory interface behaviour.

B.2. Automatic Gathering

To compile the machine description information, it is recommended to use `likwid_auto_bench.py`. This tool will use `likwid-topology` to gather the `model name`, `model type`, `sockets`, `cores per socket` and `threads per core`, as well as `level`, `size per group`, `group`, `cores per group` and `threads per group` for each memory hierarchy level.

The collection of benchmark information is especially tedious to compile, thus it is also automated in `likwid_auto_bench.py`, which in turn makes use of `likwid-bench` in order to take care of some of the involved quirks. `likwid_auto_bench.py` will run five typical streaming benchmarks (load, copy, update, triad and daxpy) on all memory levels, as well as with all possible core and thread counts. The results are collected in the `benchmarks` dictionary, alongside with information about the benchmark kernels (number of read and write streams, amount of data read and written per iteration and number of FLOPs required per iteration). `likwid_auto_bench.py` also takes traffic originating from write-allocation into account in the resulting bandwidths.

B.3. Manual Gathering

The user is left with a few specs to gather from the documentation, namely: `clock`, `FLOPs per cycle`, `micro-architecture`, `icc architecture flags`, `cacheline size`, `overlapping ports` and `non -overlapping ports`, as well as `cycles per cacheline transfer` and `bandwidth` for each cache-level.

C

ASM BLOCK MARKING FOR IACA

In order to make use of IACA, it is necessary to mark the inner-loop block and not the remainder or peeled loop versions. `kerncraft` does this by compiling the source to its assembly representation and then using the `iaca_marker.py` tool (also part of kerncraft) to identify and mark the inner-loop block. There are multiple options available for the user to influence this behaviour: automatic selection (default), manual selection (`--asm-block=manual`) or selection by block index (`--asm-block=<block index>`).

Blocks are identified by searching for conditional jump instructions referencing the last label in the assembly. The region of interest is reduced by `kerncraft`, by including `nop` instructions just before and after the kernel loops, in order to exclude initialization loops from the search.

The default and automatic selection is based on the assumption that the inner-loop block will contain the most packed (or vectorized) instructions, because no other loop is allowed to contain any arithmetic statements (see Section 3.2.1) and the remainder and peel loops are meant to prepare the data for the optimized loop.

If this were to fail, e.g., if scalar code is used, the user can use the manual mode to interactively select the correct block. `kerncraft` will present the collected statistics for each block (total number of instructions, packed instructions, AVX instructions, number of used registers, grouped by class, and the loop counter increment found at the end of the loop block) and allow the user to decide which block to mark. See Listing C.1 for a typical report from a 2D-5pt kernel, compiled with AVX.

```
Blocks found in assembly file:
   block    | OPs | pck. | AVX || Registers |    YMM    |    XMM    |     GP    || l.inc |
------------+-----+------+-----++-----------+-----------+-----------+-----------++-------|
 0 L_B1.36 |   9 |    0 |   0 ||  24 ( 12) |  0 ( 0) | 11 ( 6) | 13 ( 6) ||      1 |
 1 L_B1.39 |  55 |   16 |  16 || 175 ( 32) | 84 (15) | 16 (11) | 75 ( 6) ||     16 |
 2 L_B1.43 |   9 |    0 |   0 ||  24 ( 12) |  0 ( 0) | 11 ( 6) | 13 ( 6) ||      1 |
Choose block to be marked [1]:
```

Listing C.1: Manual ASM block selection statistics and interface

In case of batch processing, it is useful to select the same block automatically, without interaction. This is accomplished by passing `--asm-block` the index reported in the left most column of the statistics table.