Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels

Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein

Abstract Achieving optimal program performance requires deep insight into the interaction between hardware and software. For software developers without an in-depth background in computer architecture, understanding and fully utilizing modern architectures is close to impossible. Analytic loop performance modeling is a useful way to understand the relevant bottlenecks of code execution based on simple machine models. The Roofline Model and the Execution-Cache-Memory (ECM) model are proven approaches to performance modeling of loop nests. In comparison to the Roofline model, the ECM model can also describes the single-core performance and saturation behavior on a multicore chip.
We give an introduction to the Roofline and ECM models, and to stencil performance modeling using layer conditions (LC). We then present Kerncraft, a tool that can automatically construct Roofline and ECM models for loop nests by performing the required code, data transfer, and LC analysis. The layer condition analysis allows to predict optimal spatial blocking factors for loop nests. Together with the models it enables an ab-initio estimate of the potential benefits of loop blocking optimizations and of useful block sizes. In cases where LC analysis is not easily possible, Kerncraft supports a cache simulator as a fallback option. Using a 25-point long-range stencil we demonstrate the usefulness and predictive power of the Kerncraft tool.

Julian Hammer
Erlangen Regional Computing Center, Germany, e-mail: julian.hammer@fau.de

Jan Eitzinger
Erlangen Regional Computing Center, Germany, e-mail: jan.eitzinger@fau.de

Georg Hager
Erlangen Regional Computing Center, Germany, e-mail: georg.hager@fau.de

Gerhard Wellein
Erlangen Regional Computing Center, Germany, e-mail: gerhard.wellein@fau.de

1 Introduction

Expensive, large-scale supercomputers consisting of thousands of nodes make performance a major issue for efficient resource utilization. A lot of research in this area concentrates on massive scalability, but there is just as much potential for optimization at the core and chip levels. If performance fails to be acceptable at small scales, scaling up will waste resources even if the parallel efficiency is good. Therefore, performance engineering should always start with solid insight at the smallest scale: the core. Using this approach will give the performance engineer a profound understanding of performance behavior, guide optimization attempts and, finally, drive scaling at the relevant hardware bottlenecks.

Modeling techniques are essential to understand performance on a single core due to the complexities hidden in modern CPU and node architectures. Without a model it is hardly possible to navigate through the multitude of potential performance bottlenecks such as memory bandwidth, execution unit throughput, decoder throughput, cache latency, TLB misses or even OS jitter, which may or may not be relevant to the specific application at hand. Analytic models, if applied correctly, help us focus on the most relevant factors and allow validation of the gained insights. With "analytic" we mean models that were derived not by automated fitting of parameters of a highly generic predictor function, but by consciously selecting key factors that can be explained and understood by experts and then constructing a simplified machine and execution model from them.

We understand that the application of analytic performance modeling techniques often poses challenges or tends to be tedious, even for experienced software developers with a deep understanding of computer architecture and performance engineering. Kerncraft [7], our tool for automatic performance modeling, addresses these issues. Since its first publication, Kerncraft has been throughly extended with the layer condition model, an independent and more versatile cache simulation, as well as more flexible support for data accesses and kernel codes. These enhancements will be detailed in the following sections. Kerncraft is available for download under GPLv3 [1].

1.1 Related Work

Out of the many performance modeling tools that rely on hardware metrics, statistical methods, curve fitting, and machine learning, there are only four projects in the area of automatic and analytic modeling that we know of: PBound, ExaSAT, Roofline Model Toolkit and MAQAO.

Narayanan et al. [13] describe a tool (PBound) for automatically extracting relevant information about execution resources (arithmetic operations, loads and stores) from source code. They do not, however, consider cache effects and parallel execution, and their machine model is rather idealized. Unat et al. [20] introduce the ExaSAT tool, which uses compiler technology for source code analysis and also em-

ploys "layer conditions" [17] to assess the real data traffic for every memory hierarchy level based on cache and problem sizes. They use an abstract simplified machine model, whereas our Kerncraft tool employs Intel IACA to generate more accurate in-core predictions. On the one hand this (currently) restricts Kerncraft's in-core predictions to Intel CPUs, but on the other hand provides predictions from the actual machine code containing all compiler optimizationsr. Furthermore, ExaSAT is restricted to the Roofline model for performance prediction. Being compiler-based, ExaSAT supports full-application modeling and code optimizations, which is work in progress for Kerncraft. It can also incorporate communication (i.e., message passing) overhead, which is not the scope of our research. Lo et al. [14] introduced in 2014 the "Empirical Roofline Toolkit," (ERT) which aims at automatically generating hardware descriptions for Roofline analysis. They do not support automatic topology detection and their use of compiler-generated loops introduces an element of uncertainty. Djoudi et al. [2] started the MAQAO Project in 2005, which uses static analysis to predict in-core execution time and combines it with dynamic analysis to assess the overall code quality. It was originally developed for the Itanium 2 processor but has since been adapted for recent Intel64 architectures and the Xeon Phi. As with Kerncraft, MAQAO currently supports only Intel architectures. The memory access analysis is based on dynamic run-time data, i.e., it requires the code to be run on the target architecture.

1.2 Performance Models

Performance modeling, historically done by pen, paper and brain, has a long tradition in computer science. For instance, the well-known Roofline model has its origins in the 1980s [8]. In this paper, we make use of the Roofline and the Execution-Cache-Memory (ECM) models, both of which are based on a bottleneck analysis under a throughput assumption. Detailed explanations of the models can be found in previous publications; we will limit ourselves to a short overview.

1.2.1 Roofline

The Roofline model yields an absolute upper performance bound for a loop. It is based on the assumption that either the data transfers to and from a single level in the memory hierarchy or the computational work dictates the runtime. This implies that all data transfers to all memory hierarchy levels perfectly overlap with each other and with the execution of instructions in the core, which is too optimistic in the general case. The Roofline model in the current form was popularized and named by Williams et al. in 2009 [21].

For the types of analysis Kerncraft supports it is useful to reformulate the Roofline model in terms of execution time instead of performance, and to use a basic unit of work that spans the length of a cache line (typically eight iterations):

Table 1 Overview of data transfers and bandwidths necessary to model a 3D seven-point stencil kernel using the Roofline model.

Level k	Data Volume per 8 It. β_k	STREAM copy Bandwidth B_k	Time for 8 It. T_k
L1	448 B (only LOAD)	137.1 GB/s	9.8 cy
L2	7 CL or 384 B	68.4 GB/s	16.6 cy
L3	5 CL or 256 B	38.8 GB/s	24.7 cy
MEM	3 CL or 128 B	17.9 GB/s	32.2 cy

$T_{\text{roof}} = \max_k (T_{\text{core}}, T_k)$. The ratio $T_k = \beta_k / B_k$, with the achievable peak bandwidth B_k and data transfer volume β_k, is the data transfer time for memory hierarchy level k. $T_{\text{core}} = \phi / P_{\max}$ is the in-core execution time for computations with the amount of work ϕ. The latter is usually given in flops, but any other well-defined metric will do. $P_{\max}$ is the applicable computational peak performance (in flops per cy) of the code at hand. It may be smaller than the absolute peak performance because of unbalanced multiply and add operations, because SIMD cannot be applied, etc.

Applying the Roofline model to a loop kernel which loads 448 bytes from the first level cache (L1), 6 cache lines (CL) from the second level cache (L2), 4 CLs from the last level cache (L3), and two CLs from main memory, to produce one cache line of results (8 iterations), gives us the data volumes in Table 1. This is what we would expect with a 3D seven-point stencil (see Listing 1) for a certain problem size that leads to a 3D-layer condition fulfilled in L3 and a 2D-layer condition fulfilled in L2 (see below for more on layer conditions). For the computational work, we assume 5 additions and 7 multiplications per iteration, thus 96 FLOPs for eight iterations, i.e., $\phi = 96\,\text{flop}$. Combining this with measured bandwidths from a STREAM [15] copy kernel on an Ivy Bridge EP processor in all memory hierarchy levels, we can derive the throughput time per eight iterations shown in the last column of Table 1. The achievable peak bandwidth B_k is obtained via a streaming benchmark since theoretical bandwidths published by vendors cannot be obtained in practice. The ECM model provides a partial explanation for this effect, so it requires less measured input data (see below).

The double precision maximum applicable performance of a code with $5/7$ addition-multiplication ratio on an Ivy Bridge core is

$$P_{\max} = \frac{40\,\text{flop}}{7\,\text{cy}}$$

which yields an in-core prediction of

$$T_{\text{core}} = \frac{96\,\text{flop}}{40\,\text{flop}/7\,\text{cy}} = 16.8\,\text{cy}$$

The dominating bottleneck is therefore the transfer from main memory T_{MEM} with 32.2 cy for eight iterations or updates, which corresponds to a maximum expected ("lightspeed") performance of 8.94 Gflop/s.

Predicting the L1 time and performance with the measured bandwidth can only be precise if the microbenchmark mimics exactly the load/store ratio as found in the modeled code. To circumvent this issue it is advisable to use a static analyzer with knowledge of the architecture, like the Intel Architecture Core Analyzer (IACA) [10]. It also allows a more accurate prediction of T_{core}.

1.2.2 Execution-Cache-Memory

The Execution-Memory-Cache (ECM) model is based on the same fundamental idea as the Roofline model, i.e., that data transfer time or execution of instructions, whichever takes longer, determine the runtime of a loop. Unlike in the Roofline model, all memory hierarchy levels contribute to a single bottleneck. Depending on the microarchitecture, data transfer times to different memory hierarchy levels may overlap (as in the Roofline model) or they may add up. This latter assumption was shown to fit measurements quite well on x86-based processors [17, 22]; on Power8, for instance, the cache hierarchy shows considerable overlap [9]. In the following we will concentrate on Intel architectures, since the current version of Kerncraft implements a strict non-overlapping ECM model.

We also need to know the data volumes transferred to and from each memory hierarchy level and the amount of work performed in the core. To calculate the time contributions per cache level we use documented inter-cache throughputs (e.g., two cycles per cache line from L3 to L2 on Intel Ivy Bridge). The ECM prediction on an Intel core for data in memory is then given by

$$T_{\text{ECM,Mem}} = \max\left(T_{\text{OL}}, T_{\text{nOL}} + T_{\text{L1-L2}} + T_{\text{L2-L3}} + T_{\text{L3-MEM}}\right) \ .$$

T_{OL} is the overlapping time for computations and stores, T_{nOL} is the time for the loads from registers into L1, T_{L1L2} the loads from L2 into L1, and so on. The model is written in the following compact notation:

$$\left\{T_{\text{OL}} \,\|\, T_{\text{nOL}} \,|\, T_{\text{L1-L2}} \,|\, T_{\text{L2-L3}} \,|\, T_{\text{L3-MEM}}\right\} \ .$$

See [17] for more details on the model and the notation.

Applying the ECM model to the 3D seven-point stencil (see Listing 1) on an Ivy Bridge EP processor, we get the in-core contributions from IACA:

$$T_{\text{OL}} = 13.2\,\text{cy} \quad \text{and} \quad T_{\text{nOL}} = \beta_{\text{L1}} \cdot 1\frac{\text{cy}}{64\,\text{B}} = 7\,\text{cy} \ .$$

The data transfers through the memory hierarchy are obtained from cache simulation in combination with hardware performance characteristics:

$$T_{\text{L1-L2}} = \beta_{\text{L2}} \cdot 2\frac{\text{cy}}{\text{CL}} = 14\,\text{cy}$$

$$T_{\text{L2-L3}} = \beta_{\text{L3}} \cdot 2\frac{\text{cy}}{\text{CL}} = 10\,\text{cy}$$

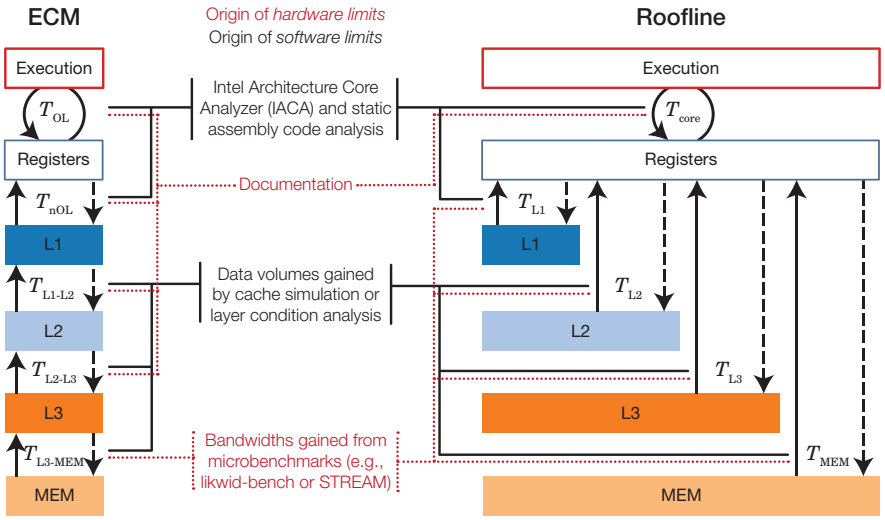

Fig. 1 Side-by-side comparison of the (x86) ECM model and the Roofline model, including the origin of information needed as input for both, such as bandwidth and execution bottlenecks.

$$T_{\mathrm{L3-MEM}} = \frac{\beta_{\mathrm{MEM}} \cdot 3.0\frac{\mathrm{Gcy}}{\mathrm{s}} \cdot 64\frac{\mathrm{B}}{\mathrm{CL}}}{63.4\frac{\mathrm{GB}}{\mathrm{s}}} = 9.1\,\mathrm{cy}$$

The ECM notation for eight iterations of the 3D seven-point stencil code is then:

$$\{13.2 \,\|\, 7 \,|\, 14 \,|\, 10 \,|\, 9.1\}\ \mathrm{cy}\ .$$

A comparison of the data that goes into the ECM and Roofline analysis (manual and automatic) is shown in Figure 1. It also illustrates the fundamental differences in the bottleneck assumption.

2 Kerncraft

In this section we give an overview of the architecture and analysis modes available in Kerncraft. The recent additions, which have not been covered in our 2015 publication [7], will be explained in due detail.

The core of Kerncraft is responsible for parsing and extracting information from a given kernel code, abstracting information about the machine, and providing a homogenous user interface. The modules responsible for the modeling will be described in Section 2.3. A visualization of the overall structure is shown in Figure 2. The user has to provide a kernel code (described in Sec. 2.1) and a machine description (described in Sec. 2.2), and they have to select a performance model to apply (options are described in Sec.2.3). Optionally, parameters can be passed to the ker-

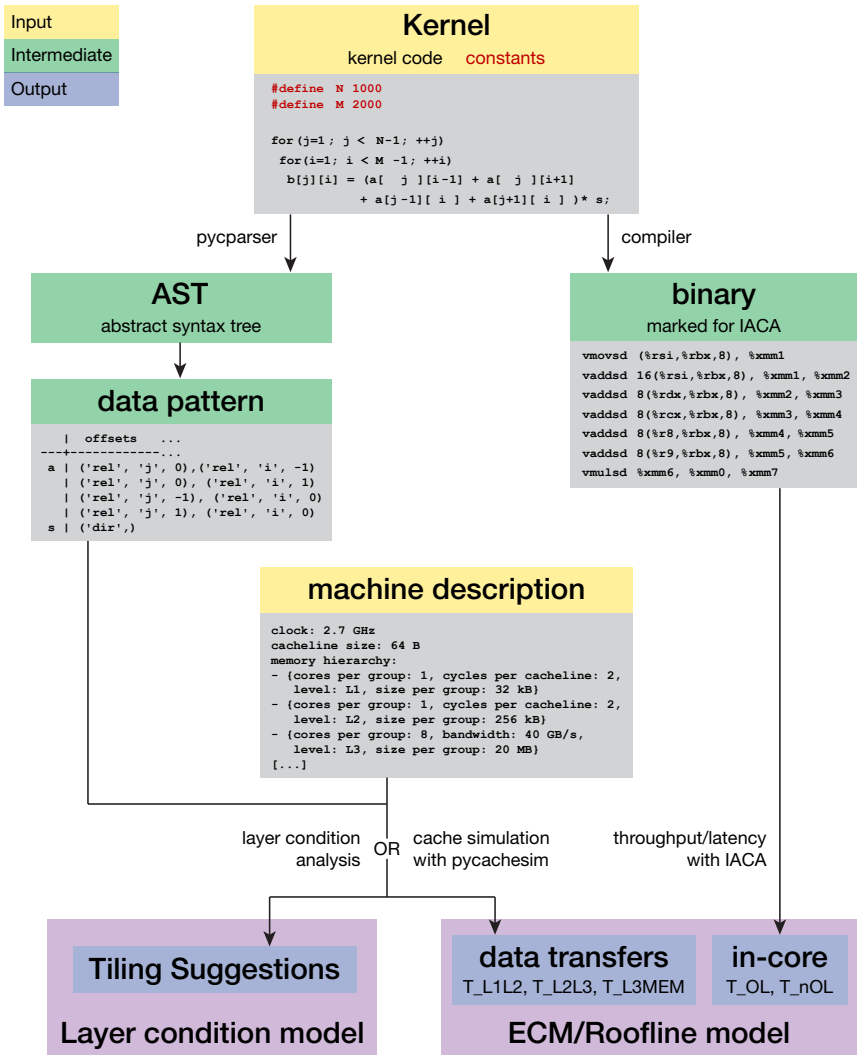

Fig. 2 Overview of the Kerncraft pipeline. The user provides kernel code, constants, and a machine description. IACA, pycachesim, and a compiler are employed to build the ECM, Roofline, and layer condition models.

nel code, similar to constants defined by macros or `-D` compiler flags. For models that rely on prediction of caching, either the layer condition prediction or the cache simulation (using the pycachesim module) can be employed. Both predictors will be described in Section 2.4.

Listing 1 Input kernel code for a three-dimensional 7-point star stencil (3D-7pt).

```
double a[M][N][N];
double b[M][N][N];
double coeffs_N, coeffs_S, coeffs_W, coeffs_E,
       coeffs_F, coeffs_B, 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] = ( coeffs_W*a[k][j][i-1]
                         + coeffs_E*a[k][j][i+1]
                         + coeffs_N*a[k][j-1][i]
                         + coeffs_S*a[k][j+1][i]
                         + coeffs_B*a[k-1][j][i]
                         + coeffs_F*a[k+1][j][i]) * s;
```

2.1 Kernel Code

Kerncraft is based on the analysis of standard-compliant C99 [11] code, which must be provided as shown in Listing 1. Example files for several stencils are distributed with the Kerncraft repository[1]. The first lines are dedicated to variable and array definitions. While large arrays would in practice be allocated on the heap, Kerncraft requires arrays to be declared as local varaibles. The multidimensional syntax (e.g., `a[M][N]` and `a[j][i]`) is optional, since Kerncraft now also supports flattened indices (e.g., `a[M*N]` and `a[j*N+i]`).

`N` and `M` in Listing 1, are constants which can be passed to the code through the command line. During analysis they are treated as symbols, which may be replaced by constant positive integers.

Following the variable definitions is the loop nest, which may only contain one loop per level and only the innermost loop may contain variable assignments and arithmetic operations. The loop indices must be local to that loop and the bounds may only depend on constant integers and simple arithmetic operations (addition, subtraction, and multiplication) of constant integers. The step size can be any constant length; in Listing 1 we have a step size of one, but `k+=2` would for instance also work.

Any number of statements are allowed in the loop body, as long as they are assignments and arithmetic operations based on constants, integers, variables, and array references. Array references may contain arithmetic expressions in their indices (e.g., `a[j*N+i+1]`). Such an expression may only be composed of constants, integers, and loop index variables (`i`, `j`, and `k` in Listing 1).

Function calls, `if`s, pointer arithmetic, and irregular data accesses are not allowed, since they could not be analyzed with the algorithms used in the current

[1] https://github.com/RRZE-HPC/kerncraft/tree/master/examples/kernels

version of Kerncraft. Moreover, the underlying models do not yet have a canonical way of dealing with the effects arising in such cases.

2.2 Machine Description

To select the targeted machine architecture, Kerncraft needs a machine description file in the YAML file format [3]. Example machines description files are distributed through the Kerncraft repository[2]. A machine description file always consists of three parts: the execution architecture description, the cache and memory hierarchy description, and benchmark results of typical streaming kernels. In the following, we will go into some settings found in Listing 2 that are not self-explanatory.

Compute Architecture

The first section is the execution architecture description (the actual order of elements does not matter in the YAML file format). This section describes the compute capabilities of the machine, such as clock speed, number of cores, or compiler flags to use for benchmarks. `micro-architecture` is the abbreviation for the Intel microarchitecture codename as used by IACA (e.g., `HSW` for Haswell), `overlapping-ports` are the execution ports corresponding to the overlapping portion in the ECM model as reported by IACA, `non-overlapping-ports` are all other ports as reported by IACA.

The machine description file with the benchmark section and partial information about the memory hierarchy and compute architecture can be generated automatically by the script `likwid_bench_auto.py`, which comes with Kerncraft. It employs `likwid-topology` and `likwid-bench` [19] to gather accurate information about the machine it is executed on.

Memory Hierarchy

Each level of the memory hierarchy has an entry in the `memory hierarchy` section. `cores per group` is the number of physical cores that share one resource on this level (e.g., if every core has its own private cache, `cores per group` is 1). `threads per group` is the number of virtual threads that share one resource on this level. `groups` is the total number of resources of this type (e.g., an L1 cache) that exist on all sockets. `cycles per cacheline transfer` is only needed for caches, except for the last level cache (LLC). It denotes the number of cycles it takes to load one cache line from the adjacent "lower" (closer to main

[2] `https://github.com/RRZE-HPC/kerncraft/tree/master/examples/machine-files`

Listing 2 Shortened machine description for Haswell (skipped sections are marked by . . .).

```
# Execution Architecture:
model name: Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
micro-architecture: HSW
non-overlapping ports: [2D, 3D]
overlapping ports: ['0', 0DV, '1', '2', '3', '4', '5', '6', '7']
FLOPs per cycle:
  DP: {ADD: 8, FMA: 8, MUL: 8, total: 16}
  SP: {ADD: 16, FMA: 16, MUL: 16, total: 32}
compiler: icc
compiler flags: [-O3, -xAVX, -fno-alias]
...
# Memory and Cache Hierarchy:
memory hierarchy:
    - level: L1
      cache per group: {
         'sets': 64, 'ways': 8, 'cl_size': 64, # 32 kB
         'replacement_policy': 'LRU',
         'write_allocate': True, 'write_back': True,
         'load_from': 'L2', 'store_to': 'L2'}
      cores per group: 1
      threads per group: 2
      groups: 28
      cycles per cacheline transfer: 2
    ...
# Benchmark Description and Results:
benchmarks:
  kernels:
    copy:
      FLOPs per iteration: 0
      read streams: {bytes: 8.00 B, streams: 1}
      read+write streams: {bytes: 0.00 B, streams: 0}
      write streams: {bytes: 8.00 B, streams: 1}
    ...
  measurements:
    L1:
      1:
        cores: [1, 2, 3, ...]
        results:
          copy: [36.15 GB/s, 72.32 GB/s, 108.48 GB/s, ...]
          ...
        size per core: [21.12 kB, 21.12 kB, 21.12 kB, ...]
        size per thread: [21.12 kB, 21.12 kB, 21.12 kB, ...]
        threads: [1, 2, 3, ...]
        threads per core: 1
        total size: [21.12 kB, 42.24 kB, 63.36 kB, ...]
    ...
```

memory) cache. For the last level cache this number is calculated from the measured saturated memory bandwidth.

The `cache per group` dictionary contains the cache description as required by pycachesim [6]. `write_back` makes sure that a modified cache line is transferred to the `store_to` cache in case of its replacement. `write_allocate` enforces a load of the cache line if some part of it is updated. The product of `sets`, `ways`, and `cl_size` is the size of one cache resource in bytes.

Benchmarks

Streaming benchmark results are required input for the Roofline model with all core counts and in all memory hierarchy levels. The ECM model only needs the measured saturated main memory bandwidth. In order to cover the whole memory hierarchy and typical effects and configurations, many tests are performed and their results stored in the machine description file. First, all benchmark kernels need to be specified in the `kernels` dictionary. For each kernel, `FLOPs per iteration` is the number of floating-point operations per iteration of the underlying kernel. `read streams` is the number of bytes and different streams read at each iteration. The ratio `bytes/streams` is the size of one element in the processed array. `read+write streams` are accesses that are both read and written to (e.g., `a` in `a[i] = a[i] + 1`). `write streams` complements `read streams`. The differentiation into these three metrics is important to handle write-allocate transfers correctly.

The benchmark results are then grouped into memory hierarchy levels and SMT threads. Each such block has the configuration per physical core, with measured bandwidth (without write-allocate), used memory size (total, per thread and per core), and the number of cores and threads used.

2.3 Models

The models offered in Kerncraft are: `Roofline`, `ECM`, `Layer Conditions`, and `Benchmark`. Although not all are, strictly speaking, performance models, each one allows some unique and valuable insight into the performance, or some aspect of expected behavior, of the kernel at hand.

Roofline

The Roofline model is implemented in the two variants `Roofline` and `RooflineIACA`. The former counts flops in the high level code and matches them to the `FLOPs per cycle` configuration in the machine description file. It also models the first level cache to register transfers using the corresponding measured band-

width result. `RooflineIACA`, on the other hand, uses the IACA analysis to predict in-core or compute performance and first level cache to register throughput. This analysis will be explained in detail in the ECM section below.

Apart from the differences in the in-core and first level cache to registers bottlenecks, both variants use the same approach for analysis throughout the rest of the memory hierarchy: take the cache miss prediction (explained in Section 2.4) and predict the required data volume (β_k) coming out of each memory hierarchy level per iteration. Take these volumes and divide them by the measured achievable bandwidths (B_k) out of the corresponding hierarchy level, which yields a throughput time for that data amount ($T_k = \beta_k / B_k$). Out of the numerous benchmarks (as described in Section 2.2), Kerncraft tries to find the one matching the kernel under examination as closely as possible with regard to the number of read and write streams into memory.

If IACA is available and a supported Intel architecture is analyzed, the `RooflineIACA` model is to be preferred over the regular `Roofline` model, as it will yield a much better accuracy.

ECM

Three versions of the Execution-Cache-Memory (ECM) model are supported: `ECMData` (modeling only the first level cache to main memory data transfers), `ECMCPU` (modeling only the in-core performance and first level cache to registers) and `ECM` (combining the data and in-core predictions). `ECMPCPU` relies on a suitable compiler and IACA to be available, which is why the rest of the ECM model can be run separately.

`ECMData` uses either the layer conditions or cache simulation (both explained in Section 2.2) to predict the data volumes out of each memory hierarchy level. Then it applies the documented bandwidths for inter-cache transfers and the measured full-socket main memory bandwidth for the memory to LLC transfers. By taking the ratio of data volume and bandwidth, T_{L1L2}, T_{L2L3}, and T_{L3Mem} are calculated (on machines with three cache levels). The benchmark kernel used for the main memory bandwidth is chosen according to the read and write stream counts best matching the analyzed kernel.

`ECMCPU` requires that the kernel is analyzed by IACA, which in turn requires a compilable version of the kernel. The kernel code is therefore wrapped in a `main` function that takes care of initializing all arrays and variables. Dummy function calls are inserted to prevent the compiler from removing seemingly useless data accesses. Once compiled to assembly language using appropriate optimizing flags, the innermost kernel loop is extracted and the unrolling factor is determined from it. Both are done using heuristics and may fail; if they do, interaction by the user is requested. Using the unrolling factor, the IACA predictions can be scaled to iterations in the high-level kernel code. IACA reports throughput cycle counts per port, which are then accumulated into T_{nOL} and T_{OL} based on the machine description configuration.

Layer Conditions

To predict optimal blocking sizes, layer conditions can be formulated in an algebraic way and solved for block sizes. The details are explained in [5], while the concept of layer conditions and our generic formulation is described in Section 2.4.2.

Benchmark

To allow validation of the previously explained models, the benchmark model compiles and runs the code and measures performance. The code is prepared in basically the same way as for an IACA analysis, but arrays are initialized and LIKWID marker calls are inserted to enable precise measurements using hardware performance counters. The output of `likwid-perfctr` is used to derive familiar metrics (Gflop/s, MLUP/s, etc.), which in turn are used for validations. It is important to note that this model must be executed on the same machine as the one in the machine description file passed to Kerncraft, otherwise results will not be conclusive.

2.4 Cache Miss Prediction

One of the core capabilities of Kerncraft is the prediction of the origin of data within the memory hierarchy, which can currently be done via two methods: a partial cache simulation using pycachesim, or a layer condition analysis. Both prediction methods have their strong points and drawbacks. Cache simulation can capture some irregularities arising from the cache structure and implementation in hardware (such as associativity conflicts) and at the same time is more generic and versatile in terms of architectural features and the kernels it can be used for. Layer conditions, on the other hand, yield very clean and stable results without disturbance from hardware-specific issues. They can be evaluated very quickly and almost independently of the code and domain size, but they only work for least-recently-used (LRU) replacement policies and currently only handle sequential traversal patterns.

In summary, if the layer condition prediction can be applied to the kernel and architecture of interest, it is usually the better choice.

2.4.1 Cache Simulation with pycachesim

The open source pycachesim library is a spin-off from Kerncraft. It is designed to efficiently model all the common cache architectures found in Intel, AMD, and Nvidia products.[3] The cache architecture is described in the machine description file and then modeled in pycachesim. It supports inclusive and exclusive caching, multiple

[3] Kerncraft currently only supports Intel Xeon and Core architectures, but pycachesim has been developed with other architectures in mind.

replacement policies (LRU, RR, Random and FIFO) as well as victim caches. For the Intel architectures covered in this paper, inclusive write-back caches with LRU are assumed. The simulator, once initialized with the cache structure, gets passed accessed data locations (loads and stores), which are followed through the simulated memory hierarchy. It also keeps a statistic about accumulated load, store, hit, and miss counts. After a warm-up phase, the statistic is reset, data accesses from a precise number of loop iterations are passed to the simulator, and the updated statistic is read out. The gained information reflects the steady state behavior.

It is very important to align the end of the warm-up period with cache line boundaries, as well as with edges of the arrays to skip over boundary handling (e.g., loops that go from 2 to $N-3$). If these cases are not considered, imprecise and oscillating performance predictions are likely.

2.4.2 Layer Conditions

Another approach to predicting the cache traffic are the Layer Conditions [16, 17]. In order to utilize them for our purposes, we have generalized and reformulated them to allow symbolic evaluation. The symbolic evaluation heavily relies on sympy [18], a computer algebra system for python.

The basis of layer conditions is the least-recently-used replacement policy, which (although typically not perfectly implemented in large, real caches) mimics observed behavior quite well. By taking the relative data access offsets and assuming sequential increments during the subsequent iterations, we can predict very precisely which access will hit or miss depending on given cache sizes.

For demonstration we assume a double precision 2D 5-point stencil on $\mathrm{M} \times \mathrm{N}$ arrays `a[M][N]` and `b[M][N]`, with accesses in the *j*th and *i*th iteration to `a[j-1][i]`, `a[j][i-1]` `a[j][i+1]`, `a[j+1][i]` and `b[j][i]`. The inner loop index is `i`. Now we compute the offsets between all accesses after sorting them in increasing order (as already shown), e.g., `&a[j][i-1] - &a[j-1][i]` or $(N-1)$ elements. We store them in the list L and insert, per array, another ∞, since we do not know the offsets between the arrays:

$$L = \{ \underbrace{\infty}_{\substack{\text{first access}\\ \text{to a}}} , \underbrace{N-1}_{\substack{\texttt{\&a[j][i-1]}\\ \texttt{- \&a[j-1][i]}}} , \underbrace{2}_{\substack{\texttt{\&a[j][i+1]}\\ \texttt{- \&a[j][i-1]}}} , \underbrace{N-1}_{\substack{\texttt{\&a[j+1][i]}\\ \texttt{- \&a[j][i+1]}}} , \underbrace{\infty}_{\substack{\text{first access}\\ \text{to b}}} \}$$

For each reuse distance t in L we can derive the required cache size C_{req}, hits C_{hits}, and misses C_{misses}:

$$\begin{aligned} C_{\mathrm{req}} &= \sum(L_{\leq t}) + t * \mathrm{count}(L_{>t}) \\ C_{\mathrm{hits}} &= \mathrm{count}(L_{\leq t}) \\ C_{\mathrm{misses}} &= \mathrm{count}(L_{>t}) \ . \end{aligned}$$

Here, $L_{\mathrm{condition}}$ is a sublist of L that contains only entries that fulfill the given condition (e.g., $L_{<t}$ contains all elements out of L which are smaller than t). Applying this

method to the described kernel, we have the interesting case $t = N-1$, for which we get $C_{\text{req}} = 2(N-1)+2+2(N-1) = 4N-2$ elements, or $32N-16$ bytes, $C_{\text{hits}} = 3$, and $C_{\text{misses}} = 2$.

This means that if an LRU-based cache can hold more than $32N-16$ bytes, three hits will be observed in each iteration and two misses will need to be passed to the next level in the memory hierarchy, which is to leading order exactly the result from a manual LC analysis (where the 16 bytes are typically neglected so that four layers, i.e., rows, must fit into the cache). Since caches in modern CPUs do not operate on bytes but on cache lines, the computed hits and misses are averaged. Once a cache line was loaded due to a miss, subsequent accesses will be hits, which averages out to the misses and hits per iteration yielded by the layer condition analysis.

2.5 Underlying In-Core Execution Prediction

To predict the in-core execution behavior, we employ the Intel Architecture Core Analyzer (IACA) [10], which predicts the throughput and latency for a sequence of assembly instructions under the assumption that all loads can be served by the first level cache. IACA presupposes steady-state execution, i.e., the loop body is assumed to be executed often enough to amortize any start-up effects.

Kerncraft operates on high level C code, which can not be analyzed by IACA directly. Therefore it first needs to be transformed into a compilable version by wrapping the kernel in a `main` function. It is then passed through a compiler and converted to assembly. The assembly sequence of the inner loop body needs to be marked to be recognized by IACA. The marked assembly is then fed into the assembler to produce an object file as input to IACA. IACA reports the throughput and latency analysis itemized by execution ports. We are interested in the overall and load-related throughput and latency. Which execution ports are associated with loads is defined in the machine description file (see Section 2.2 above). The compiler might have unrolled the inner-most loop a number of times (e.g., to allow vectorization), so this factor needs to be extracted from the assembly to scale the IACA results to a single high-level kernel code loop iteration. The IACA output is parsed and the data is presented by Kerncraft as part of the analysis.

3 Kerncraft Usage

Kerncraft guides performance engineering efforts by allowing developers to predict and validate performance. In the following sections we will use an instructive example to demonstrate the single-core performance prediction, the scaling from single-core to the full socket, and the analytic layer conditions. The analysis will be based on the long-range 3D kernel (3d-long-range) in Listing 3. Predictions and

Listing 3 Kernel code for a three dimensional long-range star stencil with constant coefficients.

```
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;
}}}
```

Table 2 Technical data of the Ivy Bridge-based node used for the long-range stencil case study.

Microarchitecture	Ivy Bridge EP
Abbreviation	IVY
Model Name	E5-2690v2
Clock (fixed, no turbo)	3.0 GHz
Cores per socket	10
Cacheline size	64 B
Theoretical L1-L2 bandwidth	0.5 CL/cy
Theoretical L2-L3 bandwidth per core	0.5 CL/cy
Achievable single-socket memory bandwidth (copy kernel)	47.2 GB/s (7 cores)
Compiler version	Intel ICC 16.0.3
IACA version	2.1
Kerncraft version	0.4.3

measurements will be done for the Intel Ivy Bridge EP (IVY) microarchitecture. The details of the machine are described in Table 3.

Listing 4 Excerpt from the kerncraft CLI (reformatted for brevity) for the analysis of the long-range stencil

```
$ kerncraft -p ECM -p RooflineIACA --cache-predictor=SIM \
            3d-long-range.c -m IVY.yaml -D M 130 -D N 1015;
============================ kerncraft ============================
3d-long-range-stencil.c                                  -m IVY.yaml
-D M 130 -D N 1015
------------------------------ ECM -------------------------------
{ 52.0 || 54.0 | 40.0 | 24.0 | 48.5 } cy/CL
{ 54.0 \ 94.0 \ 118.0 \ 166.5 } cy/CL
saturating at 4 cores

------------------------ RooflineIACA --------------------------
Bottlenecks:
 level | a. intensity | performance   | bandwidth  | bw kernel
-------+--------------+---------------+------------+----------
   CPU |              | 18.22 GFLOP/s |            |
    L2 |  0.26 FLOP/B | 17.52 GFLOP/s | 68.37 GB/s | copy
    L3 |  0.43 FLOP/B | 16.57 GFLOP/s | 38.79 GB/s | copy
   MEM |  0.43 FLOP/B |  7.65 GFLOP/s | 17.91 GB/s | copy

Cache or mem bound with 1 core(s)
7.65 GFLOP/s due to MEM bottleneck (bw with from copy benchmark)
Arithmetic Intensity: 0.43 FLOP/B
```

3.1 Single-Core Performance

Using Kerncraft for a single-core performance analysis involves choosing an overall prediction model (ECM or Roofline) and a cache predictor model (pycachesim simulation [SIM] or layer conditions [LC]). An example using `RooflineIACA`, `ECM`, and `SIM` is shown in Listing 4. It is easy to do parameter studies via simple scripting, and scanning a range of problem sizes often leads to valuable insights. Running this analysis from $N = 100$ to $N = 2000$, we can see the effect of the inner dimension increasing and visualize it in Figure 3.

The ECM prediction (stacked areas from $T_{\mathrm{nOL}} + T_{\mathrm{L1-L2}} + T_{\mathrm{L2-L3}} + T_{\mathrm{L3-MEM}}$) follows the trend of the measured throughput (black plus signs). The Roofline prediction (green dashed line) is generally too optimistic due to the evenly distributed runtime contribution from multiple memory hierarchy levels, which is not correctly modeled in this particular case. The cache simulator, taking the associativity of all cache levels into account, correctly identifies L1 thrashing and a corresponding runtime increase near $N = 1792 = 7 \cdot 256$. The corresponding increase in traffic between L1 and L2 of more than 50% can be shown using performance counter measurements. Many more such "pathological" sizes exist, of course, but the size range was not scanned with a step size of one. In Figure 4 the same parameter study was done with the LC predictor. Since it knows nothing about cache organization, the prediction is much smoother.

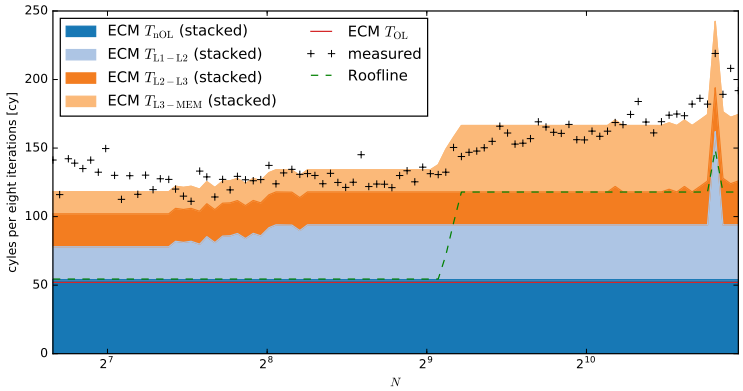

Fig. 3 Single-core parameter sweep of the long-range stencil for $N = 100$ to $N = 2000$ with M chosen such that the working set will never fit into any cache and needs to be loaded from main memory.

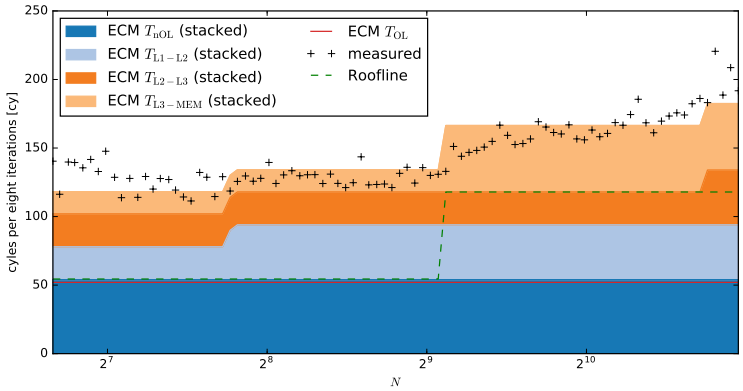

Fig. 4 Single-core parameter sweep, with layer condition cache prediction, of the long-rang stencil for $N = 100$ to $N = 2000$ with M chosen such that the data will never fit into any cache and needs to be loaded from main memory.

3.2 Single-Socket Scaling and Saturation Point

For multi-core scaling the ECM model assumes perfect scalability until a shared bandwidth bottleneck (usually the main memory bandwidth) is hit. It thus predicts the number of cores where the loop performance ceases to scale:

$$n_{\mathrm{s}} = \frac{T_{\mathrm{ECM,Mem}}}{T_{\mathrm{L3-Mem}}} \quad .$$

By default, Kerncraft reports the saturation point in the ECM model, as seen in Listing 4. The default report assumes that the total cache size and cache bandwidth scales with the number of cores. This is mostly true on current Intel microarchitectures, but not for the last level cache (L3) size, which is shared among all cores in a socket. To also take that change of cache sizes into account, Kerncraft can be run with the `--cores` argument. In the case presented in Listing 4, a reduction of the L3 cache size by a factor of four (for 4 cores) does not change the predicted results, since no layer condition changes.

To perform the single-socket scaling we added OpenMP pragmas to the outer loop in the code and ran with the same problem size as seen in Listing 4 (strong scaling). The result can be seen in Figure 5: By increasing the number of cores up to the predicted saturation point (four cores), we expect perfect scaling (dashed gray line), and constant performance beyond (dotted line). The scaling model fits the observations very well except right before the saturation point, which is a known weakness of the ECM model with data-bound kernels [17].

3.3 Layer Conditions

Layer conditions enable a much more efficient cache behavior prediction without extensive parameter studies through the simulator or benchmarks. As explained in Section 2.4.2, they are evaluated analytically and yield a prediction for transition points from one cache state to another. Kerncraft generally employs analytic LCs when using the option `--cache-predictor=LC`, but it can also output the de-

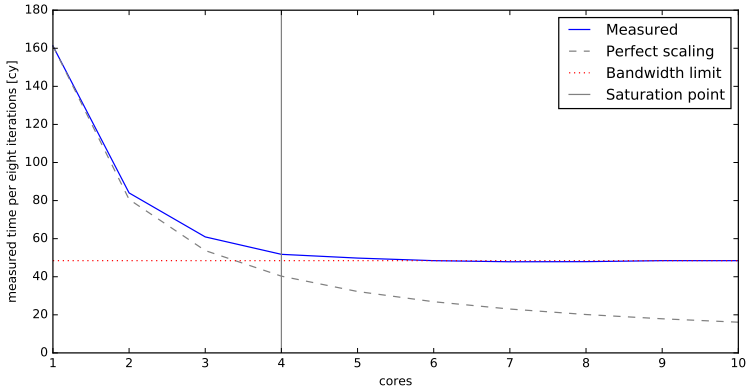

Fig. 5 Single-socket strong scaling of the long-range stencil for $N = 1015$ and $M = 132$ with all cores on same socket. The vertical line denotes the predicted saturation point. The horizontal line is the minumum runtime as given by the saturated memory bandwidth.

Listing 5 Excerpt from the kerncraft CLI (reformatted for brevity) showing LC transition points from the analysis of the long-range stencil

```
$ kerncraft -p LC 3d-long-range.c -m IVY.yaml -D M 130 -D N 1015;
============================ kerncraft ============================
3d-long-range-stencil.c                                  -m IVY.yaml
-D M 130 -D N 1015
------------------------------ LC -------------------------------
2D Layer-Condition:
L1: N <= 216
L2: N <= 1725
L3: N <= 172463
3D Layer-Condition:
L1: N <= 19
L2: N <= 55
L3: N <= 546
```

rived transition points as shown in Listing 5. The predicted transition in L3 from the 3D to the 2D layer condition at $N = 546$ is also clearly visible in Figures 3 and 4.

4 Future Work

Development on Kerncraft will continue and strive to enhance usability and portability and to allow support of a broader range of kernels and architectures. One of the major obstacles to supporting non-Intel CPUs is IACA, which is closed-source and only supports Intel microarchitectures. It is our goal to develop a model and tool which will be suitable for predictions on other architectures. In the near future we will also integrate our layer condition model with the LLVM-Polly project [4]. This will allow the Polyhedral model to automatically choose cache-efficient tiling sizes without user interaction.

As with all of our tools and libraries (Kerncraft, LIKWID [19], GHOST [12], and the soon-to-be-published fault-tolerance package CRAFT), future work will be released under open source licenses and we will support and encourage other projects to build upon them.

Acknowledgements This work was in part funded by the German Academic Exchange Service's (DAAD) FITweltweit program and the Federal Ministry of Education and Research (BMBF) SKAMPY grant.

References

[1] Kerncraft toolkit. https://github.com/RRZE-HPC/kerncraft

[2] Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.T., Jalby, W., et al.: MAQAO: Modular assembler quality analyzer and optimizer for itanium 2. In: The 4th Workshop on EPIC architectures and compiler technology, San Jose (2005). http://www.prism.uvsq.fr/users/bad/Research/ps/maqao.pdf

[3] Evans, C., Ingerson, B., Ben-Kiki, O.: YAML Ain't Markup Language. http://yaml.org

[4] Grosser, T., Groesslinger, A., Lengauer, C.: Polly – performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(04), 1250,010 (2012). DOI 10.1142/S0129626412500107

[5] Hammer, J.: Layer conditions. URL https://rrze-hpc.github.io/layer-condition/

[6] Hammer, J.: pycachesim – a single-core cache hierarchy simulator written in python. URL https://github.com/RRZE-HPC/pycachesim

[7] Hammer, J., Hager, G., Eitzinger, J., Wellein, G.: Automatic loop kernel analysis and performance modeling with kerncraft. In: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, PMBS '15, pp. 4:1–4:11. ACM, New York, NY, USA (2015). DOI 10.1145/2832087.2832092

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

[9] Hofmann, J., Fey, D., Riedmann, M., Eitzinger, J., Hager, G., Wellein, G.: Performance analysis of the Kahan-enhanced scalar product on current multicore and many-core processors. Concurrency and Computation: Practice and Experience pp. n/a–n/a (2016). DOI 10.1002/cpe.3921

[10] Intel Architecture Code Analyzer. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[11] ISO: ISO C Standard 1999. Tech. rep. (1999). URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. ISO/IEC 9899:1999 draft

[12] Kreutzer, M., Thies, J., Röhrig-Zöllner, M., Pieper, A., Shahzad, F., Galgon, M., Basermann, A., Fehske, H., Hager, G., Wellein, G.: GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems. International Journal of Parallel Programming pp. 1–27 (2016). DOI 10.1007/s10766-016-0464-z

[13] Krishna Narayanan, S.H., Norris, B., Hovland, P.D.: Generating performance bounds from source code. In: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, pp. 197–206 (2010). DOI 10.1109/ICPPW.2010.37

[14] Lo, Y., Williams, S., Van Straalen, B., Ligocki, T., Cordery, M., Wright, N., Hall, M., Oliker, L.: Roofline model toolkit: A practical tool for architectural and program analysis. In: S.A. Jarvis, S.A. Wright, S.D. Hammond (eds.) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, *Lecture Notes in Computer Science*, vol. 8966, pp. 129–148. Springer International Publishing (2015). DOI 10.1007/978-3-319-17248-4_7. DOI: 10.1007/978-3-319-17248-4_7

[15] McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, VA (1991-2007). URL `http://www.cs.virginia.edu/stream/`. A continually updated technical report

[16] Rivera, G., Tseng, C.W.: Tiling optimizations for 3D scientific computations. In: Supercomputing, ACM/IEEE 2000 Conference, pp. 32–32 (2000). DOI 10.1109/SC.2000.10015

[17] Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM International Conference on Supercomputing, ICS '15, pp. 207–216. ACM, New York, NY, USA (2015). DOI 10.1145/2751205.2751240

[18] SymPy Development Team: SymPy: Python library for symbolic mathematics (2016). URL `http://www.sympy.org`

[19] Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures. San Diego CA (2010)

[20] Unat, D., Chan, C., Zhang, W., Williams, S., Bachan, J., Bell, J., Shalf, J.: ExaSAT: An exascale co-design tool for performance modeling. International Journal of High Performance Computing Applications **29**(2), 209–232 (2015). DOI 10.1177/1094342014568690. DOI: 10.1177/1094342014568690

[21] Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). DOI 10.1145/1498765.1498785

[22] Wittmann, M., Hager, G., Zeiser, T., Treibig, J., Wellein, G.: Chip-level and multi-node analysis of energy-optimized lattice Boltzmann CFD simulations. Concurrency and Computation: Practice and Experience **28**(7), 2295–2315 (2016). DOI 10.1002/cpe.3489