YaskSite: Stencil Optimization Techniques Applied to Explicit ODE Methods on Modern Architectures

Christie L. Alappat
University of Erlangen-Nuremberg
Erlangen, Germany
christie.alappat@fau.de

Johannes Seiferth
University of Bayreuth
Bayreuth, Germany
johannes.seiferth@uni-bayreuth.de

Georg Hager
University of Erlangen-Nuremberg
Erlangen, Germany
georg.hager@fau.de

Matthias Korch
University of Bayreuth
Bayreuth, Germany
matthias.korch@uni-bayreuth.de

Thomas Rauber
University of Bayreuth
Bayreuth, Germany
thomas.rauber@uni-bayreuth.de

Gerhard Wellein
University of Erlangen-Nuremberg
Erlangen, Germany
gerhard.wellein@fau.de

***Abstract*—The landscape of multi-core architectures is growing more complex and diverse. Optimal application performance tuning parameters can vary widely across CPUs, and finding them in a possibly multidimensional parameter search space can be time consuming, expensive and potentially infeasible. In this work, we introduce YaskSite, a tool capable of tackling these challenges for stencil computations. YaskSite is built upon Intel's YASK framework. It combines YASK's flexibility to deal with different target architectures with the Execution-Cache-Memory performance model, which enables identifying optimal performance parameters analytically without the need to run the code. Further we show that YaskSite's features can be exploited by external tuning frameworks to reliably select the most efficient kernel(s) for the application at hand. To demonstrate this, we integrate YaskSite into Offsite, an offline tuner for explicit ordinary differential equation methods, and show that the generated performance predictions are reliable and accurate, leading to considerable performance gains at minimal code generation time and autotuning costs on the latest Intel Cascade Lake and AMD Rome CPUs.**

***Index Terms*—Performance modeling; ECM model; Stencil optimization; YASK; Autotuning; PIRK methods**

I. Introduction

The efficiency of applications from scientific computing is in general strongly dependent on characteristics of the targeted hardware platform and application-specific parameters. To achieve a high efficiency, such applications need to be adapted to the specific hardware platform anew. Typically, this adaptation process includes *(i)* applying program optimization techniques, such as loop blocking or loop unrolling, to the heavy-workload kernels of the application and *(ii)* selecting suitable parameter values for these optimizations, such as loop unrolling factors or block sizes. Commonly, these optimizations are employed using code generation techniques; autotuning is used to select the optimal parameter values. However, often there is a large search space of possible optimization variants with corresponding parameter values within which the most efficient configuration(s) need to be determined. In a landscape of modern multi-core architectures that are getting more diverse, implementing and testing all these variants for each new architecture is cumbersome and time consuming, especially when done by hand.

In this work, we introduce YaskSite, a tool capable of automating this complex adaptation process for stencil-based computations. YaskSite can generate stencil codes with various optimizations and can automatically tune these codes using an analytical performance model. This enables YaskSite to produce near-optimal stencil/streaming code with practically no tuning overhead, as would be incurred in standard runtime testing.

A. Related Work

An established solution for automating the identification of efficient code variant(s) is **autotuning** (AT). In recent years, many AT approaches have been proposed for different application areas [1]. In general, AT techniques are roughly divided into two groups: Online AT approaches, such as *Active Harmony* [2] and *ATF* [3], are applied at runtime—when all input is known—which allows to consider all influences of the input data during variant selection. In contrast, Offline AT selects the supposedly most efficient variant(s) at compile/installation time and is thus well suited when the execution behavior does not depend on input data. For example, dense linear algebra problems can be tuned offline with *ATLAS* [4] and *PhiPAC* [5]. The data independence of stencil computations allows us to tune them offline in this work.

Since a major challenge in AT is identifying efficient variants from the potentially large search space, many solutions have been proposed [1]. This includes brute-force exhaustive search and selective search space scans using optimization methods like hill climbing [2], genetic algorithms [6] or hierarchical approaches as in [7]. Another approach is filtering out inefficient variants using performance models. These models describe the interaction of code and hardware using simplified application and machine models. We follow this strategy as its effectiveness for stencil and streaming codes has been demonstrated [8]. Two established analytic models for steady-state loop codes are the *Roofline model* [9] and

the *Execution-Cache-Memory (ECM) performance model* [10]. Both models are applicable to the memory-bound kernels considered in this work. The ECM model, however, requires far less phenomenological input, so we favored it in this work. *Kerncraft* [11] can provide ECM predictions for stencil and streaming kernels but does not support the optimizations considered in this work.

AT has to be supported with the generation of optimized code. Intense research has been conducted for stencil algorithms in this regard. The majority of optimizations focus on reducing the data traffic via spatial and temporal tiling [12], [13], [14], [15], [16], [17]. For long-range stencils, work on improving the in-core performance by SIMD vectorization has also been carried out [18], [19]. Frameworks like YASK [19], PATUS [16] or Girih [20] support some of these optimizations along with AT. Most of them are based on well-known approaches like genetic algorithms and hill-climbing. In this work we use the YASK framework as the backend.

B. Main Contributions

The main contributions of this paper are:

(i) We propose an analytical tuner for stencil computations (*YaskSite*), which links YASK with an analytic performance model based on the ECM model. We demonstrate that YaskSite is capable of analytically and automatically detecting near-optimal optimization parameters without requiring any runtime tests and compare it with YASK's built-in tuner.

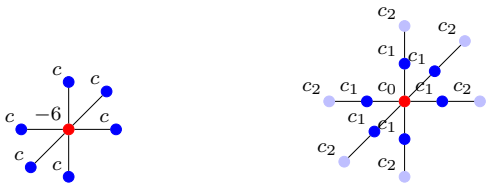
(ii) We provide a performance model for YASK's stencil code that considers spatial tiling and vector folding optimizations. To this end, we provide a revised performance model for spatial tiling that includes victim caches and we introduce an analytical performance model for vector folding, which to our knowledge is the first analysis of this kind. The model is further verified for two stencil algorithms, *Heat* and *Wave*, on *Intel Cascade Lake* (CLX) and *AMD Rome* (ROME).

(iii) We demonstrate that the insights gained from the performance model can guide code optimization by improving the performance of a specific YASK-generated stencil by a factor of 2.5 on ROME.

(iv) We integrate YaskSite into an existing offline autotuning framework for explicit ODE (PIRK) methods (*Offsite*). In particular, we expand Offsite to incorporate YaskSite-optimized stencil code variants and integrate YaskSite's performance model to estimate and reliably rank the performance of these variants.

C. Outline

Section II introduces the selected example use case (PIRK methods) and Section III details our testbed. In Section IV, we introduce YASK and discuss how YaskSite expands YASK by integrating performance modeling. Section V describes YaskSite's performance modeling workflow. In Section VI, we give an overview of the Offsite autotuning approach and discuss how YaskSite's features are integrated to expand Offsite's scope of application. Section VII studies the accuracy of the model by comparing performance predictions for PIRK method implementations to measurements in different application scenarios on different target platforms. Finally, Section VIII and Section IX summarize and conclude the paper.



(a) Heat3D. (b) Wave3D ($r = 2$).

Fig. 1: Characteristic stencil of ODE problems *Heat3D* and *Wave3D*.

II. Case Study: PIRK Methods

We study *parallel iterated Runge-Kutta (PIRK) methods* [21] as a representative example of the general class of explicit ODE methods. PIRK methods solve ODE systems

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \ , \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \ , \tag{1}$$

for an integration interval $[t_0,\ t_e]$ by performing a series of time steps $t_\kappa = t_0 + \kappa \cdot h$ until the end of the integration interval (t_e) is reached. Here, $\mathbf{f}$ is the right-hand-side function (RHS), $\mathbf{y}_0$ is the initial value of the ODE system and h is the step size.

A. Mathematical Background

PIRK methods are one-step methods, i.e., in each time step t_κ a new approximation $\mathbf{y}_{\kappa+1}$ for the unknown solution $\mathbf{y}$ is computed from the previous approximation $\mathbf{y}_\kappa$ only. PIRK methods use an *explicit predictor–corrector* process with a fixed number of $m = p-1$ corrections, where p is the order of the s-stage implicit RK method used as *corrector method*. The input vector $\mathbf{y}_\kappa$ of the time step is chosen as initial approximation (*predictor*) for the stages $\mathbf{Y}_1, \ldots, \mathbf{Y}_s$:

$$\mathbf{Y}_l^{(0)} = \mathbf{y}_\kappa, \quad l = 1, \ldots, s \quad . \tag{2}$$

Next, the corrector method of order p with its Butcher table entries, i.e. coefficient matrix $A = (a_{ij}) \in \mathbb{R}^{s,s}$, weight vector $\mathbf{b} = (b_i) \in \mathbb{R}^s$ and node vector $\mathbf{c} = (c_i) \in \mathbb{R}^s$, is applied $m = p - 1$ times, i.e. for $k = 1, \ldots, m$:

$$\mathbf{Y}_l^{(k)} = \mathbf{y}_\kappa + h_\kappa \sum\nolimits_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}, \quad l = 1, \ldots, s \tag{3a}$$

with

$$\mathbf{F}_i^{(k-1)} = \mathbf{f}\left(t_\kappa + c_i h, \mathbf{Y}_i^{(k-1)}\right) \ . \tag{3b}$$

After all corrector steps are computed,

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum\nolimits_{i=1}^{s} b_i \mathbf{F}_i^{(m)} \tag{4}$$

yields the output approximation $\mathbf{y}_{\kappa+1}$.

In this work, we use *Radau II A(7)* with $s = 4$ and $p = 7$ ($m = 6$) as corrector method.

```
//BARRIER
for(int k=0; k<m; ++k){
  RHS
  //BARRIER
  LC
  //BARRIER
}
RHS
//BARRIER
App
//BARRIER
Up
```

a: Implementation *A*.

```
//BARRIER
for(int k=0; k<m; ++k){
  RHS_LC
  //BARRIER
}
RHS_App_Up
```

b: Implementation *F*.

Listing 1: Schemes of two PIRK method implementations. Implicit OpenMP barriers are shown for reference.

B. ODE Problems

We study two ODE problems whose stencil-like RHS allows to explore YaskSite's stencil optimization opportunities on fundamental equations for various scientific fields:

(i) Heat3D is a 3D heat equation with Dirichlet boundary conditions and describes the temperature distribution in a given volume over time. Its RHS is characterized by a seven-point stencil (Fig. 1a) with radius $r = 1$ on a cubic grid ($N = \sqrt[3]{n}$) and a problem-dependent coefficient $c \in \mathbb{R}$.

(ii) Wave3D[1] is a 3D wave equation with Dirichlet boundary conditions, which describes, e.g., the propagation of seismic waves. For a radius r *Wave3D* stencil the pressure at the next time step $t+1$ is given by:

$$\begin{aligned} y^{t+1}_{x,y,z} = 2 \cdot y^{t}_{x,y,z} - y^{t-1}_{x,y,z} + v_{x,y,z} \cdot (c_0 \cdot y^{t}_{x,y,z} \\ + \sum_{i=1}^{r} c_i \cdot (y^{t}_{x-i,y,z} + y^{t}_{x+i,y,z} + y^{t}_{x,y-i,z} \\ + y^{t}_{x,y+i,z} + y^{t}_{x,y,z-i} + y^{t}_{x,y,z+i})) \end{aligned} \tag{5}$$

where $y^{t}_{x,y,z}$ and $y^{t-1}_{x,y,z}$ are the pressure at the current and previous time steps and v is the velocity vector. The coefficients of the finite differences on a cubic grid ($N = \sqrt[3]{n}$) are given by $c_i \in \mathbb{R}, i \in \{1, \ldots, r\}$. The RHS is characterized by a $(6r+1)$-point stencil corresponding to the y^t vector (Fig. 1b).

C. PIRK Solvers

We consider two selected shared-memory implementations for which the predictability of their performance by the ECM model has been discussed before [8], [22]:

(i) Implementation *A* (Listing 1a) is a vector-oriented, easy to predict implementation that was thoroughly studied in [8]. It splits equations (2)–(4) into separate kernels. A single corrector iteration k (3) comprises two non-overlapping kernels: Kernel *LC* (Listing 2a) covers the linear combination (3a), while kernel *RHS* (Listing 2b) evaluates the RHS functions (3b) of the ODE problem. For kernel *LC*, there are six possible permutations of loops i, j and l. In this work, we only consider permutation j-i-l which was most performant in [8]. To compute the next approximation of the solution (4), kernels *App* (Listing 2c) and *Upd* (Listing 2d), are used.

(ii) Implementation *F* (Listing 1b) is derived from *A* by loop fusion and significantly outperformed *A* in [22]; however, it was not yet studied on ODE problems with a 2D/3D stencil-like RHS. The two kernels executed per corrector step iteration, *RHS* and *LC*, are fused into a single kernel *RHS_LC* (Listing 2e). The computation of the next solution approximation is covered by kernel *RHS_App_Up* (Listing 2f) which is derived by fusing kernels *RHS*, *App* and *Up*.

[1]Also known as *Iso3DFD* (isotropic 3D finite-difference) [6].

```
for(j•=0; j•<N; ++j•) {
  for(i=0; i<s; ++i) { // unrolled second
    for(l=0; l<s; ++l) { // unrolled first
      tmp_arr[l] += A[l][i] * F[i][jx][jy][jz]
    }
  }
  for(l=0; l<s; ++l) { // unrolled third
    Y[l][jx][jy][jz] = tmp_arr[l] * h +
        y_old[jx][jy][jz]
  }
}
```

a: Kernel *LC*.

```
for(j•=0; j•<N; ++j•)
  for(i=0; i<s; ++i) // unrolled
    F[i][jx][jy][jz] = RHS
```

b: Kernel *RHS*.

```
for(j•=0; j•<N; ++j•) {
  for(i=0; i<s; ++i) { // unrolled
    tmp = tmp + b[i] * F[i][jx][jy][jz]
  }
  dy[jx][jy][jz] = tmp
}
```

c: Kernel *App*.

```
for(j•=0; j•<N; ++j•) {
  y_new[jx][jy][jz] = y_old[jx][jy][jz] + h *
      dy[jx][jy][jz]
}
```

d: Kernel *Up*.

```
for(j•=0; j•<N; ++j•) {
  for(i=0; i<s; ++i) { // unrolled second
    tmp = RHS(...)
    for(l=0; l<s; ++l) { // unrolled first
      tmp_arr[l] += A[l][i] * tmp
    }
  }
  for(l=0; l<s; ++l) { // unrolled third
    Y[l][jx][jy][jz] = tmp_arr[l] * h +
        y_old[jx][jy][jz]
  }
}
```

e: Kernel *RHS_LC*.

```
for(j•=0; j•<N; ++j•) {
  for(l=0; l<s; ++l) { // unrolled
    tmp += b[l] * RHS(...)
  }
  y_new[jx][jy][jz] = y_old[jx][jy][jz] + h * tmp
}
```

f: Kernel *RHS_App_Up*.

Listing 2: Kernels[2] used by PIRK implementations *A* and *F*.[3]

III. Testbed

The experiments are conducted on the two latest x86 architectures from Intel and AMD, *Intel Cascade Lake* (CLX) and *AMD Rome* (ROME) (see Table I for basic hardware characteristics). For minimal variance and best results [23],

[2]The '•' in a for loop denotes that the loop has to be repeated in every spatial direction, with directions x, y and z replacing the '•'.

[3]The parallelization is carried over the $j_\bullet$ loop and the scheduling strategy is left to the library actually running the kernel, YaskSite in our case.

TABLE I
KEY SPECIFICATIONS OF TARGET PLATFORMS.

Name	CLX	ROME
Microarchitecture	Intel Cascade Lake-SP	AMD Zen2
CPU	Xeon Gold 6248	EPYC 7452
De-facto frequency	2.5 GHz	2.35 GHz
Cores	20	32
SIMD extensions	AVX-512	AVX-2
L1 cache	20×32 KiB	32×32 KiB
L2 cache	20×1 MiB	32×512 KiB
L3 cache	28 MiB	8×16 MiB
Memory configuration	6ch. DDR4-2933	8ch. DDR4-3200
Memory bandwidth		
Theoretical	140.8 Gbyte/s	204.8 Gbyte/s
Measured STREAM triad	105.4 Gbyte/s	104.2 Gbyte/s

we switch off automatic NUMA balancing, set the transparent huge pages option to "always," and fix the clock frequency to the respective base value. As the numerical kernels under investigation can saturate the memory bandwidth with one thread per core, simultaneous multithreading (SMT) is not used (although activated in the hardware). CLX is configured with one ccNUMA domain per socket (Sub-NUMA Clustering off). Figure 2 shows the STREAM triad bandwidth measured for CLX with the `likwid-bench` tool [24].

ROME has a hierarchical design with four cores constituting a compute core complex (CCX), and two CCXs forming a compute core die (CCD). CCDs are connected to the memory via an I/O die. Similar to CLX, the L1 and L2 caches are private, while the L3 cache is only shared among the four cores in a CCX. The socket is configured in "NPS1" mode [25], i.e., as a single ccNUMA domain. In contrast to CLX, this does not mean that all the cores can use the entire memory interface. The performance characteristic is similar to having four separate ccNUMA domains (one per CCD), which is reflected in the measured STREAM bandwidth (see Fig. 2). For performance modeling we thus consider the socket to comprise four "quasi-NUMA" domains. Interestingly, the bandwidth scaling across quasi-NUMA domains is not ideal (see ideal line in Fig. 2); we speculate that this might be caused by contention on the I/O die.

By design, YASK targets only one NUMA node for shared-memory parallelization. Therefore, we restrict the experiments to one ccNUMA domain of the architectures, i.e., one socket with the above configurations.

Software Environment: The *Intel compiler 19.0 update 2* is used on CLX with flags `-O3 -xCORE-AVX512` and with `-O3 -xCORE-AVX2` on ROME. Unless specified otherwise, YASK version 3.03.01 is employed and `LIKWID` 5.0.1 is used for performance counter measurements and microbenchmarking. All floating point computations are done with double precision.

IV. YASKSITE

In this work, we propose the **YaskSite** tool, which combines Intel's DSL-based toolkit for stencils, YASK, with an analytical performance model. YaskSite is capable of running optimized stencil and streaming kernels.

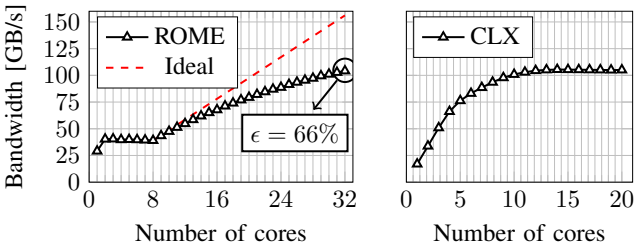


Fig. 2: Effective STREAM triad bandwidth on one socket of ROME (left) and CLX (right) versus number of cores (threads). Compact pinning across physical cores was enforced. The scaling across CCDs on ROME is not perfect and results in a final efficiency of 66%.

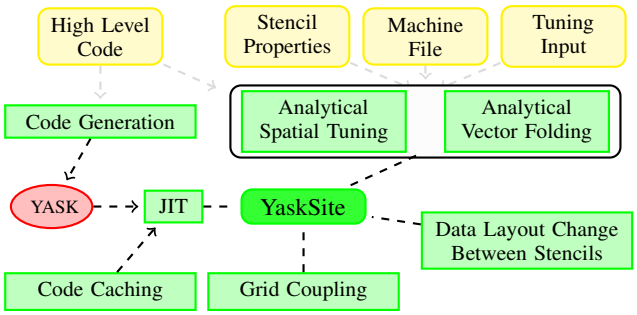


Fig. 3: Overview of the YaskSite features (green rectangles) and their interactions (edges). YaskSite inputs are shown as rounded rectangles (yellow) and external tools called by YaskSite as ellipses (red).

A. YASK

The YASK framework [6] employs a hierarchical tile structure which offers high flexibility and enables high performance on a wide variety of heterogeneous architectures. The outermost tile defines the global problem dimensions $\{g_\bullet\}$, where $\bullet$ denotes the direction (e.g., x for x-direction). The global tile is hierarchically divided into regional (r), block (b) and mini-block tiles (m). In this work, we focus only on *spatial blocking*, which means that regional and mini-block tiles are not considered. Listing 3 exemplarily shows the basic structure of a spatial blocking code generated by YASK.

The second main feature of YASK is improved code *vectorization* through *vector folding* techniques, which reduces the number of required load instructions. YASK supports folding along any dimension, with $L_\bullet$ denoting the fold length in direction $\bullet$. Besides tiling and vector folding, YASK offers automatic separation of boundaries and main grids via the bounding box concept and automatic dependency analysis between equations to fuse independent kernels. These allow to run the main body of the code without conditionals in inner kernels and to reduce the data traffic.

B. YaskSite

While offering increased flexibility and providing advanced optimization strategies in a large parameter tuning space, a major drawback is that YASK uses runtime testing steered by optimization algorithms (like genetic algorithms or gradient descent) for scanning the parameter space. This is time- and

resource-consuming. YaskSite replaces the runtime tests with an analytical performance model.

The general features of YaskSite are shown in Fig. 3. The workflow of YaskSite starts with either a YASK-compatible stencil file (in C++) or with high-level code (similar to Listing 2). For the latter, YaskSite automatically generates the YASK *stencil file*, which is processed by YASK and compiled to a shared library that can be dynamically linked to YaskSite. The entire process happens at runtime (just-in-time) whenever a new stencil is required by the application. To reduce overhead, YaskSite can optionally cache previous builds of a stencil. The performance model uses the stencil properties, generated stencil code and the machine characteristics provided in a YAML machine file to automatically predict the performance and determine the optimal stencil parameters based on the tuning input. The parameters are then set and the code is executed via YASK kernel API routines. The machine file is an integral part of the performance prediction and is the only component that needs to be generated for a new machine/setting. YaskSite uses the Kerncraft tool[4] and some additional benchmarks to automatically generate this machine file for a given hardware upon user request.

YaskSite further provides helper functions to change the data layout and to couple grids between different stencils when using multiple stencil kernels in one application; e.g., the input of kernel *RHS_App_Up* and the output of kernel *RHS_LC* should be coupled in implementation *F* (Listing 1b).

V. Performance Modeling in YaskSite

The ECM model requires the calculation of several runtime contributions, which are the numbers of cycles to carry out:

(i) in-core instructions other than loads (T_{comp})
(ii) in-core load/store instructions, i.e., L1 to Register (T_{L1})
(iii) data transfer between caches and memory, (e.g., between L2 and L1 given as T_{L2})[5].

Depending on the architecture, the data-related contributions may or may not overlap, while T_{comp} always overlaps with the others. The ECM model considers this by a machine-specific *overlap hypothesis*, which can be determined by an iterative procedure. It has been shown in [26] that this approach is compatible with wide variety of modern multi-core architectures. Similar to its predecessors, CLX has no overlap in the data transfer times, i.e., they have to be added up. On ROME, however, full overlap applies, which means that the component taking the most cycles determines the overall runtime prediction.

All contributions T_* are derived from application and machine information and are defined per lattice site update (LUP). T_{comp} and T_{L1} are determined by the instructions produced by the compiler and the in-core port model and instruction throughput of each of the ports. Tools such as the Intel Architecture Code Analyzer (IACA) [27] and the Open-Source Architecture Code Analyzer (OSACA [28], [29]) can determine both contributions via a static analysis of the object or assembly code. The remaining components (e.g., T_{L2}) require an in-depth analysis of the data transfers of the application code, which we discuss in the following Sections V-A and V-B in the context of the YASK code.

By combining the ECM model contributions, the *single-core runtime prediction* is given by:

$$T(1) = \max\left(T_{\mathrm{comp}},\ \mathrm{op1}\left(T_{\mathrm{L1}},\ \mathrm{op2}\left(T_{\mathrm{L2}},\ \mathrm{op3}(T_{\mathrm{L3}}, \dots\right)\right)\right) \tag{6}$$

where each of op1, op2, op3, ... is either SUM or MAX operator, depending on the overlap hypothesis. The multi-core prediction $T(\xi)$ within a NUMA domain is obtained by scaling down $T(1)$ by the number of cores (ξ) up to the saturation point ξ_s.[6] At CPU frequency f,

$$\xi_s = T(1)/T_{\mathrm{Mem}}$$
$$T(\xi) = T(1)/\max(\xi, \xi_s) \tag{7a}$$
$$P(\xi) = f/T(\xi) \tag{7b}$$

yields the predicted runtime $T(\xi)$ and performance $P(\xi)$ for ξ cores on a ccNUMA domain. Beyond a single domain, the model assumes linear speedup. On ROME, however, scaling is not perfect across CCDs as shown in Fig. 2. To account for this, we multiply the performance with the measured stream efficiency when scaling outside a CCD on ROME.

A. Spatial Tiling

YASK uses spatial tiles (Listing 3) to optimize performance. Here we discuss how the tile size ($b_\bullet$) affects performance by changing the data flow through the memory hierarchy.

For any streaming code whose working set fits into memory hierarchy level j, the *minimum* data volume V_{base} per lattice site update from and to hierarchy level i is:

$$V_i = V_{\mathrm{base}} = 0 \qquad \text{if } i > j \tag{8a}$$
$$V_i = V_{\mathrm{base}} = (N_r + w \cdot N_w) \cdot v \cdot d \qquad \text{else} \tag{8b}$$

where N_r is the number of grids that are only read, N_w is the number of grids that are written and d is the size of a single grid element in bytes (e.g., `sizeof(double)`). w denotes the *write-allocate factor*, which is one for victim caches and two for all other caches.[7] Factor v is the *victim cache factor*, which is one except when data fits in a victim cache (j), in which case $v = 2$ as all data must be loaded and written back. Given a data volume V_i to level i, the ECM contribution T_i is:

$$T_i = f \cdot V_i / B_i \tag{9}$$

where B_i is the effective bandwidth (in bytes/s) between level i and its next lower neighbor. The derivation is based on the assumption that the throughput of data traffic is dominant,

[4]https://github.com/RR ZE-HPC/kerncraft

[5]In [10], [26], data transfers between caches are denoted by T_{ij}, where i and j are the two memory hierarchy levels involved. For ease of notation, we use T_{j} in this work to denote transfers between cache/memory j (e.g., L2) and lower cache levels (e.g., L1).

[6]This uses the assumption that all resources except the shared ones (main memory) scale linearly.

[7]It is assumed that write-allocate avoiding techniques like non-temporal stores and/or cache line zero are not used. If they are, the factor w is one.

```
1 for(int t=1; t<g_t; ++t)
2 #pragma omp parallel for collapse(3) schedule(static,1)
3   for(int begin_b●=0; begin_b●<g_●; begin_b● += b_●)
4     for(int ● = begin_b●; ● < begin_b● + b_●; ● = ● + 1)
5       out[t+1,x,y,z] = STENCIL(in[t, x, y, z])
```

Listing 3: Basic structure of the spatial blocking code[2] considered. The remainder loop handling is not shown. The *collapse* clause refers to the three nested blocking loops in x, y and z direction covered by the loop counters "begin_b●".

while latency costs can be neglected. This is valid for the stencil kernels considered in this work due to the regular access patterns with long inner loops and hardware prefetchers being active for all our experiments. The calculation of the effective bandwidth (B_i) depends on the property of cache/memory system, i.e., the duplexity and symmetry (between load and store) of the bandwidth. More details can be found in [26].

The lowest hierarchy level j in which a data set fits entirely determines the transition point from (8a) to (8b). It is identified by comparing the total memory footprint of the code with the effective cache size[8] S_i of each memory hierarchy level (i). If $N_g = N_r + N_w$ is the total number of grids (with grid dimension $g_\bullet$), then j denotes the *lowest* cache hierarchy (i) that satisfies $N_g \cdot d \cdot g_x \cdot g_y \cdot g_z < S_i$.

Moving from pure streaming to stencil operations, additional data traffic contributions have to be taken into account. These depend on the data reuse within an update sweep of the grid. For example, data is always reused along the inner dimension (z), i.e., grid elements $(x, y, z - r)$, $(x, y, z - (r - 1))$, ..., $(x, y, z + (r - 1))$ reuse the element $(x, y, z + r)$. Along the other dimensions, however, data may or may not be reused from the cache; this depends on the amount of data loaded into a cache until a grid element in that dimension is touched again. The *layer condition analysis* (LCA) introduced in [10], [11], which assumes a cache with a *least recently used* (LRU) replacement policy, can be used to analytically estimate the extra data volume and the problem size at which reuse of data in each dimension occurs. For YASK codes (Listing 3), block dimensions ($b_\bullet$) play a crucial role in this respect; star-shaped stencils[9] with radius r (e.g. *Heat3D* and *Wave3D*) have the following reuse conditions:

$$\text{reuse in } y \Rightarrow (N_s\,(4r+1) + (N_g - N_s)) \cdot b_z d < s_i \quad \text{(10a)}$$

$$\text{reuse in } x \Rightarrow (N_s\,(2r+1) + (N_g - N_s)) \cdot b_y b_z d < s_i \quad \text{(10b)}$$

They decide whether data can be reused along y and x. N_s is the number of grids that contain stencil relations. As blocks are computed separately for each thread, the cache sizes s_i differ from the total effective cache size S_i. In general, both are related by $s_i = S_i/\xi$, where ξ is the number of working cores.

Reuse in the outermost dimension x implies reuse in all other dimensions. A cache i satisfying (10b) will only have to load element $(x + r, y, z)$ from the next higher level while all other stencil accesses will be cache hits. However, if cache i only satisfies (10a) but not (10b), there is only reuse in the y and z directions, and all $2r$ elements in the x dimension as well as the newest element in the y dimension $(x, y + r, z)$ must be loaded from the higher level. If neither condition is satisfied, there is no reuse in x and y, and $4r$ elements in the x and y dimensions along with $(x, y, z+r)$ in the z dimension will be loaded. The *minimal* data traffic V_{base} in (8) assumes one load per lattice site update from all stencil grids. This applies if the next lower cache i satisfies (10b); else, further contributions corresponding to non-reused elements (see above) must be added. This is summarized in Table II.

[8]Note that S_i includes the cache size available to all the working cores.

[9]For a discussion of different stencil shapes, we refer to [30].

TABLE II
DATA TRANSFER VOLUME OF STENCIL/STREAMING CODES. THE VOLUME V_i FROM CACHE i TO A LOWER LEVEL IS DETERMINED BY THE CACHE SIZE s_i AVAILABLE TO EACH CORE WHEN GOING THROUGH THE TABLE FROM THE TOP AND STOPPING AT THE FIRST CONDITION MET. V_{base} CORRESPONDS TO $vd \cdot (N_r + wN_w)$. FOR CASES WITHOUT VECTOR FOLDING (SECTION V-A), $r_x = r_y = r$ AND $L_x = L_y = 1$.

$s_i \leq$	V_i in byte/LUP
$(N_s(2(r_x + r_y) + 1) + (N_g - N_s))L_x L_y b_z d$	$V_{\text{base}} + 2N_s(r_x + r_y)vd$
$(N_s(2r_x + 1) + (N_g - N_s))L_x b_y b_z d$	$V_{\text{base}} + 2N_s r_x vd$
$N_g d g_x g_y g_z/\xi$	V_{base}
∞	0

Analytical tuner: Block sizes $b_\bullet$ (Listing 3), i.e., b_y and b_z can be adjusted to satisfy (10a) and (10b) for a given cache. This will decrease the data volume (V_i) from the next higher level (i), resulting in a reduced data transfer time T_i. YaskSite uses this fact for analytical tuning. The YaskSite tuner can identify block sizes $b_\bullet$ from so-called *blocking criteria*. A blocking criterion is a user-provided tuning input to YaskSite as illustrated in Fig. 3. The input specifies which cache needs to satisfy which condition of (10). For example, if the user specifies that the L3 cache need to satisfy (10b), the tuner will consider the cache size of L3 (s_i) and try to set $b_\bullet$ such that only one element $(x + r, y, z)$ of the stencil grid needs to be loaded from main memory.

The tuner further tries to make the total number of block tiles a multiple of the thread count in order to avoid load imbalance. In addition, this number has to be small to reduce the extra traffic from boundaries. To avoid performance loss due to latency and prefetching effects, the tuner selects a solution which has the minimum cut (i.e., maximum b_z) in the innermost direction (z).

Figure 4 shows the impact of a violation of (10b) by the L3 cache. The measured performance and memory data volume (in inset) of the *Wave3D* stencil are plotted against g_z on one socket (20 cores) of CLX. For comparison, the corresponding ECM prediction is plotted as dotted lines (predictions for blocked code are hidden by the $r = 1$ spatial blocking measurements). We chose $g_y = g_z$, and g_x was set to make the working set 10 GiB, which allows to solely portray the layer condition effects while ignoring cases where all data fits into a cache. The number of grids with specific properties required for the prediction can be derived from (5); for this case $N_r = 3$, $N_w = 1$ and $N_s = 1$. Variant *plain* sets $b_y = b_z = g_z$ and

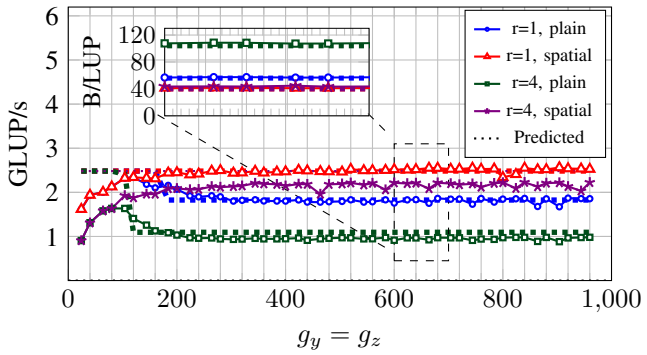


Fig. 4: Performance of *Wave3D* on CLX (full socket) vs. inner grid dimensions, with and without analytical spatial tuning at $r = 1$ and $r = 4$, comparing measurement (solid line) with ECM prediction (dotted line). Inset: memory traffic ($600 < g_z < 700$) predicted and as measured with `LIKWID`.

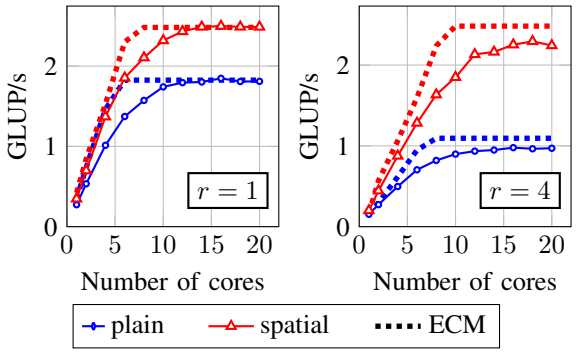


Fig. 5: Scaling performance of *Wave3D* on CLX at $g_y = g_z = 500$, with and without analytical spatial tuning at $r = 1$ and $r = 4$. The measurements are shown with solid line and ECM model prediction with dotted line.

chooses b_x to have enough work for τ working threads, i.e., $b_x = g_x/\tau$. For variant *spatial*, the YaskSite tuner selects $b_\bullet$ such that (10b) is satisfied in L3 and (10a) in L2 cache. The inset depicts for $g_z \in [600, 700]$ the memory data volume measured by `LIKWID` (solid lines) and the data volume V_{Mem} predicted (dotted lines) using V_i from Table II. Note that the performance boost by analytical blocking is higher for the larger radius ($r = 4$ vs $r = 1$) as the volume without reuse in the x-dimension has a radius-dependent term $2r$.

The scaling behavior of both the plain and spatial variants for $r = 1$ and $r = 4$ within a socket of CLX is shown in Fig. 5. The typical saturation pattern of memory-bound codes can be observed. The ECM model also captures this behavior by the use of the maximum function in (7a). The deviation between measurement and model around the saturation point is due to the decrease in the efficiency of the memory subsystem as the utilization increases. This can be corrected by adding a recursive penalty term to the model [31]. Since the aim of this work is to generate proper performance ranking and tuning among kernels, we do not follow this approach here.

Figure 6 compares YaskSite's analytical spatial tuner with YASK's inherent spatial tuning on one socket of CLX and ROME. YASK uses a runtime tuning strategy with an optimization algorithm based on gradient descent. The plots show

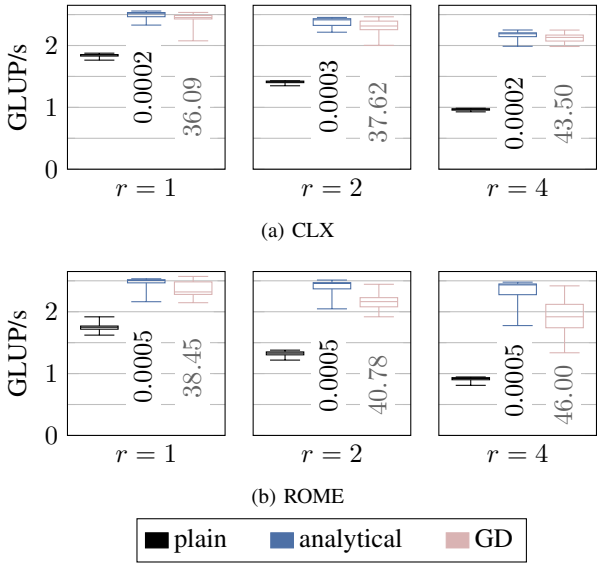


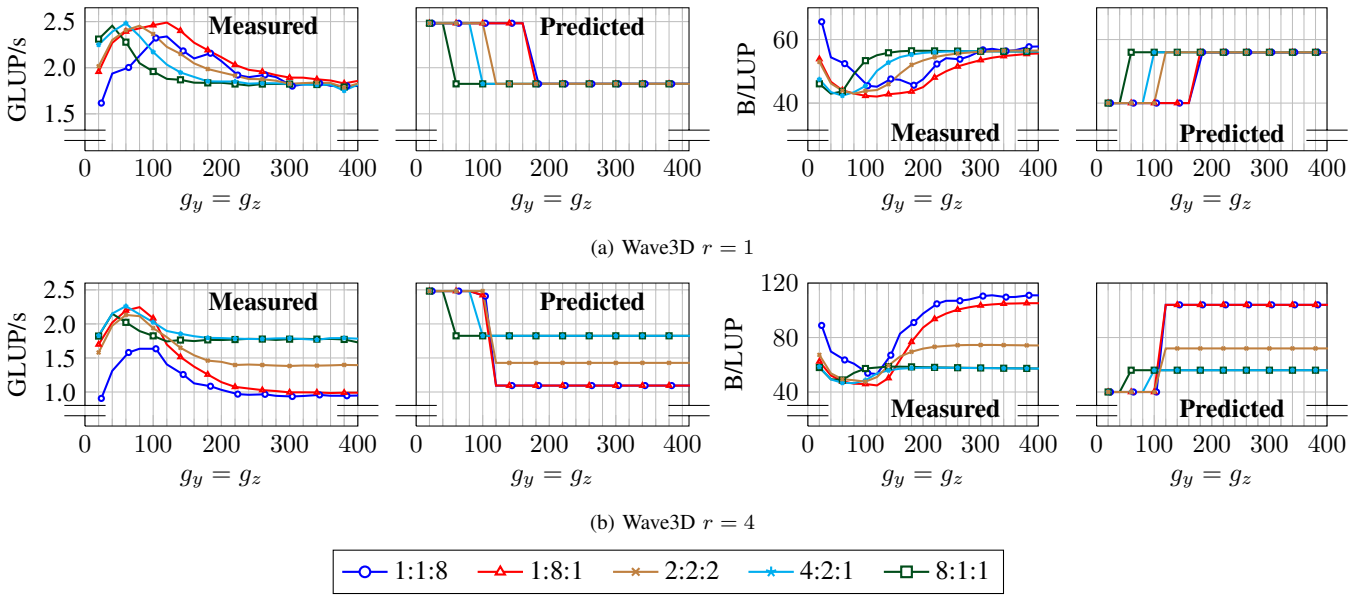
Fig. 6: Performance comparison between YaskSite's analytical spatial tuning (blue) and YASK's gradient descent (GD) tuning (light red) for *Wave3D* stencils with different radius. The plain variant (black) with no tuning is shown for reference. Numbers show the average tuning time in seconds.

the performance statistics of *Wave3D* for 36 runs with global problem dimension varying from 300 to 1000 in steps of 20. The starting block sizes ($b_\bullet$) for YASK's runtime tuning were set to 32 (default). In all cases, the parameters found by YaskSite's analytical tuner achieve a similar or better performance than YASK's tuning algorithm. Moreover, due to its analytical nature, YaskSite's tuner is very fast (average time in seconds is written in numbers in Fig. 6) and predicts the attainable performance without actually running the code. This is highly beneficial when there are many kernels and variants to test, as is the case with PIRK methods.

B. Vector Folding

Vector folding [19] was introduced in YASK as a technique to reduce L1 to register transfers, thereby reducing T_{L1}. In traditional vectorization, a separate vectorized load would be used for each stencil entry. For instance, the *Wave3D* stencil with $r = 4$ would need 25 vectorized loads (corresponding to 25 points of the stencil) of SIMD width L to update L target elements. Many of the neighboring loads in the innermost dimension are redundant, e.g., at $L = 8$, (x, y, z) is loaded five times into five different registers. By applying a 1D vector folding along the z-direction ($L_z = 8$), these redundant loads can be avoided and only three loads (compared to nine) in the z direction would suffice. To recover all nine registers that correspond to the nine points in the z-direction, blend/permute operations are carried out on the loaded vector registers, which decreases T_{L1} (load) at the price of potentially increasing the T_{comp} (blend) contributions.

The folding technique can be extended to the outer dimensions (x, y) by rearranging the data layout and moving the folded elements of the outer dimensions (L_x, L_y) to the innermost dimension. To denote the vector folding in all



(a) Wave3D $r = 1$

(b) Wave3D $r = 4$

1:1:8 1:8:1 2:2:2 4:2:1 8:1:1

Fig. 7: Performance and memory traffic of different folds for Wave3D on CLX. Note that the y-axes do not start from 0.

directions, we use the notation L_x:L_y:L_z. E.g., the 1D folding considered above is 1:1:8. By combining foldings along the dimensions, *2D folds* (e.g., 1:4:2) or *3D folds* (e.g., 2:2:2) can be constructed. Each of these requires different numbers of load and blend operations [19].

To consider folding effects in the performance model, data traffic contributions from all memory hierarchies are required. These contributions are heavily dependent on the LCA. For 1D vector foldings along the innermost direction, the LCA remains unchanged compared to Section V-A. However, for foldings in outer dimensions the LCA has to be modified to reflect the changes in the data layout.

For such foldings, two opposing effects influence the LCA. First, bringing $L_x \times L_y$ elements of the outer dimensions to the innermost z-dimension reduces the effective radius (r) in the folded dimension x and y from r to $\lceil r/L_x \rceil$ and $\lceil r/L_y \rceil$, respectively.[10] For $r > 1$, this has the positive effect of decreasing the data volume. Figure 7b depicts this for the plain variant of *Wave3D* at r=4 on one socket of CLX. With growing L_x, if there is no reuse in x-direction by the L3 cache (see, e.g., $g_z = 400$), the memory data volume decreases and correspondingly the performance increases. This happens since the extra data traffic in this case corresponds to $2r_x$, where r_x is the effective radius in the x dimension. However, for $r = 1$ (Fig. 7a) there is no difference since $\lceil 1/L_\bullet \rceil$ is always one for all folding lengths $L_\bullet$.

A second effect of the data layout change is that more data is brought into cache while traversing the z-dimension. Between two consecutive touches of the stencil elements in x- or y-direction, more data (compared to the naive version) is loaded. For example with a fold of 8:1:1, eight times more data is loaded when traversing the z-direction. This implies that the reuse conditions in y- and x-dimension ((10a) and (10b), respectively) must be modified:

$$\text{y} \Rightarrow L_x L_y \left(N_s \left(2r_x + 2r_y + 1\right) + \left(N_g - N_s\right)\right) b_z d < s_i \tag{11a}$$

$$\text{x} \Rightarrow L_x \left(N_s \left(2r_x + 1\right) + \left(N_g - N_s\right)\right) b_y b_z d < s_i \tag{11b}$$

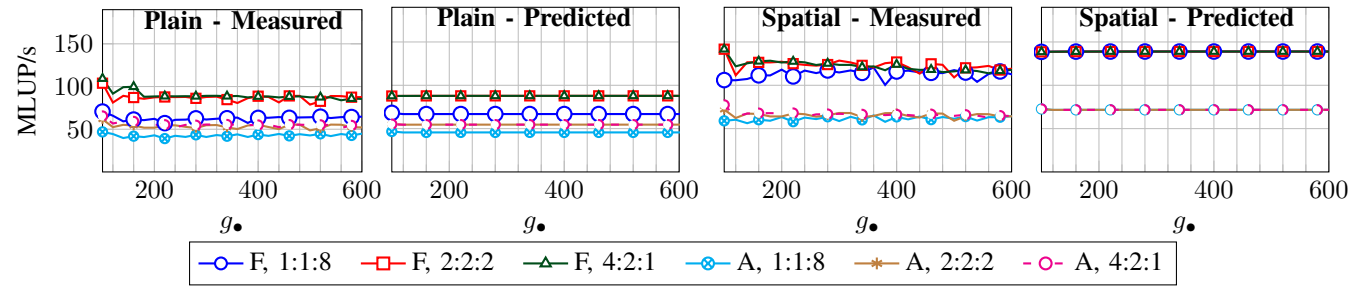
where $r_x = \lceil r/L_x \rceil$ and $r_y = \lceil r/L_y \rceil$.

Table II shows the transferred data volume (V_i) in byte/LUP. The derived V_{Mem} is plotted for *Wave3D* in Fig. 7 (two rightmost columns) together with the measured values. The dependence of L_x on the reuse in x-dimension (11b) is evident. E.g., in the case of $r = r_x = 1$, both experiment and prediction show that as L_x increases the dimension $g_z = b_y = b_z$ at which (11b) is violated shifts towards the left, i.e., smaller values. It should be noted that the drop in measured performance around the point of violation of (11b) is not as sharp compared to the prediction, leading to some deviation between model and measurement in this region. This is because the model assumes perfect LRU with full associativity, while in reality associativity is finite and pseudo-LRU is used. However, from Fig. 7 it is clear that the predicted qualitative performance behavior is in tune with the measurement and therefore the best variant can be chosen without actually generating the code, which in this case takes about 30 s for each variant. In addition, the prediction allows to save the autotuning cost as shown in Fig. 6.

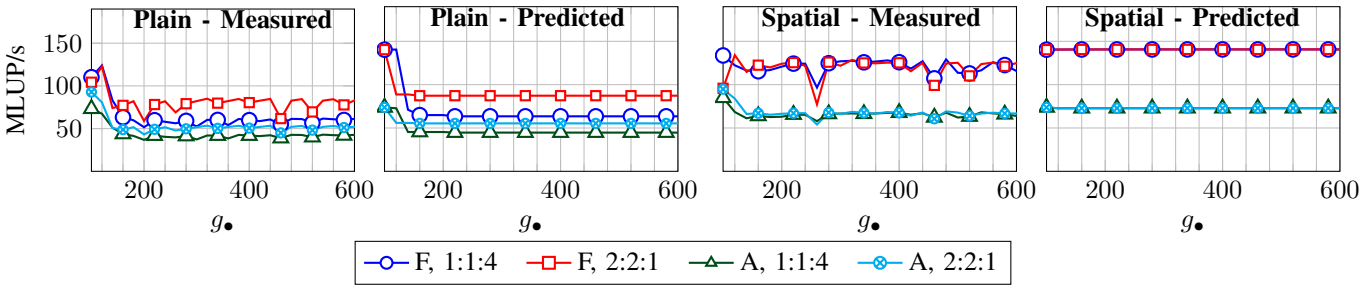
VI. Offsite Integration

To demonstrate YaskSite's features in an AT context, we integrate it into Offsite, an offline AT approach for explicit ODE methods [22]. Offsite can identify efficient implementation variants from a pool, which the user defined using YAML description formats. Offsite compares and ranks these variants

[10] $\lceil x \rceil$ denotes the smallest integer $\geq x$.



(a) Wave3D ($r = 2$) on CLX.



(b) Wave3D ($r = 2$) on ROME.

Fig. 8: Comparison of predicted and measured performance for different variants of the PIRK method using Wave3D ($r = 2$). 'F' and 'A' denote the PIRK implementation type; folds are specified with the usual L_x:L_y:L_z notation. 'Plain' and 'Spatial' refer to variants without and with analytical block size tuning, respectively.

by their performance using an analytic prediction methodology based on the ECM model [8]. By integrating YaskSite, Offsite can use its ECM predictions for blockings and foldings.

Offsite runs start with a user-defined *tuning scenario*, which includes *(i)* the pool of implementation variants, *(ii)* the target platform, *(iii)* the ODE method, and *(iv)* the ODE problem. For YaskSite, we further needed to add *(v)* parameters to specify blockings and foldings. An exemplary tuning scenario, considered later in Fig. 8a, is the following: *(i)* A and F, *(ii)* CLX *(iii)* Radau II A(7), *(iv) Wave3D* at $r = 2$ and *(v)* blocking in L3 cache for x-direction reuse with 4:2:1 folding.

Offsite derives all available variants from the given scenario and computes for each its performance prediction

$$\theta_\epsilon = \sum_{\lambda}^{|\Lambda|} \phi_\lambda + t_{\text{com}} \tag{12}$$

as the sum of the predictions ϕ of its kernels $|\Lambda|$ plus an estimate t_{com} of the barrier costs obtained via benchmarking (see [8], [32] for details). These predictions θ are used to determine the variant ranking. Offsite generates specialized code only for the best-rated variants, which is handled internally [22]. For this work, we further coupled YaskSite's code generation to enable its stencil-specific optimizations.

The prediction ϕ of a kernel λ is given as $\phi_\lambda = \frac{\beta}{P(\xi)}$, where $P(\xi)$ is the ECM prediction for λ (7b) and β is the loop iteration count. For each kernel, Offsite generates code – specialized in the input data – in a format processable by the backend used to obtain $P(\xi)$. Kerncraft is used as backend in [22]. With YaskSite, we provide an alternative backend for stencil-like codes, which allows to exploit YaskSite's optimizations within Offsite. This extends Offsite's available optimization space on blocking and folding support as well as on specialized optimization of stencil-like ODEs and, thus, considerably expands the pool of variants to select from.

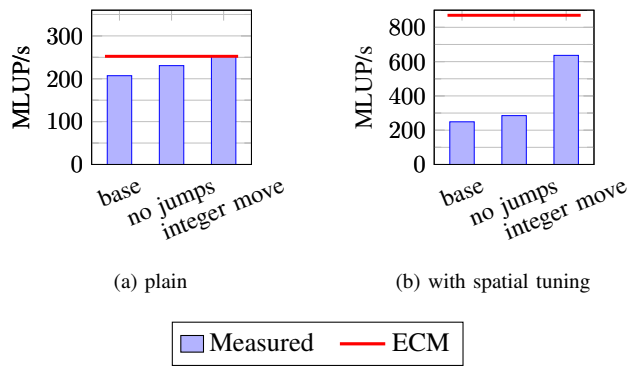
VII. Experimental Evaluation

The quality of YaskSite's performance prediction for PIRK method implementations and the reliability of the derived rankings is evaluated in the following. In particular, we study tuning ODE problems *Heat3D* and *Wave3D* (cf. Fig. 1) for ODE method *Radau II A(7)*. For *Wave3D*, we include cases $r = 2$ and $r = 4$.

For each ODE problem, 20 and 16 different implementation variants are considered on CLX and ROME, respectively. These variants are derived from PIRK implementations *A* and *F* by applying analytical spatial tuning and different vector folds ($1D$, $2D$, $3D$). The total fold size ($L_x \cdot L_y \cdot L_z$) is fixed to equal the actual hardware SIMD width, i.e., eight for CLX and four for ROME. We consider five different folds for CLX (cf. Fig. 7) and four folds (1:1:4, 1:4:1, 4:1:1, 2:2:1) for ROME. All experiments are run on cubic grids, with $g_\bullet$ varying from 120–700 in increments of 20. As the run-time variations between different measurement runs were less than 5%, we do not show the error bars in our performance results.

A. Prediction Quality

Figure 8 shows the predicted and measured performance vs. the problem size of some interesting variants for *Wave3D* (case $r = 2$) without (*Plain*) and with spatial tuning (*Spatial*) applied. Predictions and measurements agree well on both systems. Most of the deviation was at smaller problem sizes;



(a) plain (b) with spatial tuning

Fig. 9: Performance of different code optimization strategies on YASK with the *RHS_LC* kernel using the *Wave3D* ($r = 4$) stencil on ROME. The total (cubic) problem size in this case is $g_\bullet = 500$ and the folding is set to 1:1:4.

e.g., the mean deviation on ROME was 12.7% for $g_\bullet \leq 260$ and 9.0% for $g_\bullet > 260$. This higher deviation at smaller sizes is due to the extra data volume and overhead caused at the boundaries, which are currently not considered in the model.

Overall, for almost all cases the mean deviation was well under 20%, except for *Wave3D* ($r = 4$) on ROME. For *Wave3D* ($r = 4$) on ROME the mean deviation was up to 40% indicating that the generated code does not behave as predicted and might be sub-optimal. A detailed investigation of the assembly code revealed two problems, which affected kernels like *RHS_LC* (see the high deviation of the base variant in Fig. 9). Firstly, the assembly code contained conditional branches inside the main inner loop due to a generic masked store being used, although a non-masked store would be sufficient. Secondly, the main loop contained unnecessary integer move operations due to generic unaligned vectors being created despite vectors being aligned in many cases. By employing specialized versions of the instructions, we were able to fix both problems and could increase the performance of kernel *RHS_LC* compared to the base version (Fig. 9) for tuning cases "plain" and "spatial." On machines that support the AVX-512 instruction set (like CLX), the aforementioned problems did not have much impact on the performance since specialized mask and permute instructions are available. In contrast, for AVX/AVX-2 (like on ROME) YASK emulates these instructions, causing extra overhead.

Both issues were fixed in the latest YASK version 3.04.03, which was used in the experiments shown in Fig. 8. Despite all optimizations, the prediction of the tuned version of *Wave3D* ($r = 4$) was still off by about 20%. Further investigation indicated that this is due to boundary overhead and cache associativity, which can have a considerable impact on complex kernels such as *RHS_LC*.[11] To support investigations like these, YaskSite has a validation feature that measures the data traffic contributions across the memory hierarchy levels using the `LIKWID` performance counter tool and compares them with corresponding predictions. For *RHS_LC* with $r = 4$, we observed that the measured L3 traffic was more than two times higher than the analytically predicted traffic, while for other ODE problems the traffic deviation was less than 15%. These overheads and effects are not currently included in the ECM model but can be included. For instance, in the case of boundary overhead, extra data traffic proportional to the surface area of each block and the width of the halo (generally equal to radius r) can be added. Similar analyses and extensions to the model are part of our future work.

[11]The *RHS_LC* kernel includes four 25-point stencils (*Wave3D*, $r = 4$) and other streaming arrays making it a complex, bulky stencil.

TABLE III
MEAN DEVIATION AND PERFORMANCE LOSS INCURRED BY CHOOSING YASKSITE'S PERFORMANCE MODEL FOR RANKING.

Arch	IVP	Deviation	Performance loss (%)	
		Mean	Maximum	Mean
CLX	Heat3D	6.4	9.8	1.0
	Wave3D, $r = 2$	9.4	9.8	1.2
	Wave3D, $r = 4$	17.2	5.8	1.0
ROME	Heat3D	9.9	21.1	3.6
	Wave3D, $r = 2$	10.0	21.1	4.4
	Wave3D, $r = 4$	16.3	17.4	2.2

Table III summarizes for all three problems the mean performance deviation on CLX and ROME across all problem sizes and variants. Note that YASK version 3.04.03 was used in these experiments.

B. Ranking Quality

An important metric for assessing the quality of an AT is the reliability and accuracy of its returned ranking, which can be quantified using the performance loss metric [8]. It describes the loss in performance caused by executing a variant suggested by AT instead of the actually measured best variant. Ideally, both variants are the same. The performance loss is quantified as the percentage of performance deviation $\left(\frac{best-select}{select} \times 100\right)$ between the actual measured best variant (*best*) and the variant selected by AT (*select*). The lower the loss the better is the quality of AT.

Table III shows the maximum and mean performance loss of different problem sizes when using YaskSite's analytical model for ranking the variants. As primary criteria, variants are ranked by their performance prediction θ (see Sect. VI) in ascending order. In case of a tie, the variant with the lower saturation point ξ_s is ranked higher. This way the variant that is expected to saturate the memory interface first (i.e., with a lower number of cores) is selected. Table III shows that YaskSite performs very well and has a mean performance loss of well under 5% in all cases. The maximum performance loss occurred for very small problem sizes ($g_\bullet < 260$), while losses were marginal for the remaining cases. This ensures that the simple[12] analytical model considered here is sufficient to attain a proper ranking for all ODE problems and architectures considered.

[12]"Simple" refers to the simplifying assumptions like perfect LRU, full cache associativity, boundary effects being negligible, and linear scaling to saturation.

VIII. Conclusion

In this paper we have introduced YaskSite, which combines the YASK stencil code generation and autotuning toolkit with the analytical ECM performance model to automatically predict and tune the performance of stencil codes. To this end, we presented for the first time an ECM model for an AMD Rome CPU and further extended the model's layer condition analysis to include vector folding and victim caches. We showed that insights from the model can be used to analytically tune the code without actually running it and compare our results with YASK's built-in tuner. The model's usefulness to detect bottlenecks and guide performance optimizations was demonstrated via a case study on Rome, in which the sub-optimal generated YASK code could be fixed and performance improved by a factor of 2.5. Finally, we have shown that YaskSite can be integrated with external tools like Offsite to autotune more complex applications. In particular, prediction and ranking quality of different PIRK method implementation variants were analyzed for two stencil-based ODE problems and it was shown that YaskSite's predictions can reliably identify the best variants, thereby saving both code generation and autotuning costs.

IX. Outlook and Future Work

The introduced YaskSite framework allows easy expansion and testing of the built-in performance model and tuning procedures. Future work will include taking into account the overheads at boundaries, an extension to other stencil shapes (like box stencils) and adding support for temporal blocking of stencils. Another future direction of this work is to include architectures other than x86 by expanding support to other generic backends like DEVITO [33] in addition to YASK.

The presented idea to couple YASK code with an analytical performance model and tuning is not limited to stencil codes and can be generalized to other areas where analytical performance models like the ECM or Roofline model apply. These models have been widely used to model applications involving steady-state loops like sparse matrix operations with indirect accesses [34], Lattice Boltzmann Method (LBM) kernels with a large number of input arrays [35], and neuron simulations in the Blue Brain Project [36]. However, the tuning and optimization strategies will vary depending on the application field, and therefore the analytical tuner will have to be adapted to reflect the underlying optimizations and performance models.

Acknowledgment

This work is supported by the German Ministry of Science and Education (BMBF) under project numbers 01IH16012A, 01IH16012C.

Artifact Appendix

A. Abstract

This artifact provides the source code of the YaskSite framework introduced in this work. Our YaskSite framework builds on Intel's YASK framework and the analytical ECM performance model. The artifact includes a Singularity container which takes care of installing all dependencies required to build YaskSite and to run the experiments discussed in our paper.

B. Artifact Checklist

- **Program:** Running the artifact requires LIKWID (version 5.1.0) and Offsite (version 0.2.0). Both are open-source[13][14] and already included in the Singularity container.
- **Compilation:** The Intel C compiler is required. We conducted our experiments with compiler version 19.0 update 2, but other newer versions should work too.
- **Transformations:** YaskSite employs YASK version 3.04.01 to carry out required code transformations. YASK is open-source[15] and included in the Singularity container.
- **Run-time Environment:** The only supported environment is x86-64 Linux. YaskSite's dependencies include Intel YASK, Intel IACA, LIKWID, symee and yaml-cpp. Offsite requires Python 3.6 or higher. All dependencies are automatically resolved by the Singularity container.
- **Hardware:** The experiments were conducted on an Intel Cascade Lake and an AMD Rome architecture (see Table I).
- **Execution:** For minimal variance and best results [23], we switch off automatic NUMA balancing, set the transparent huge pages option to "always," and fix the clock frequency to the respective base value (see Table I). Simultaneous multithreading is not used although activated in the hardware.
- **Metrics:** The measured and predicted performance of a kernel is reported in lattice updates per second [LUP/s].
- **Output:** Yasksite writes its output to console or file. Offsite stores its outputs in a SQLite database.
- **Experiments:** Build the Singularity container and follow the instructions given in the artifact repository. For easiness, experiments in the paper can be reproduced using corresponding Singularity apps. Runtime variations between different measurement runs were less than five percent.
- **How much disk space required (approximately):** The Singularity container is approximately 2.6 GB in size. For YaskSite measurement runs approximately 60 GB main memory space are required. When tuning the PIRK application with Offsite (see Fig. 8) we require approximately 120 GB of main memory space. Moreover, 10 GB disk space is needed to cache all YASK generated kernels.
- **How much time is needed to prepare workflow?:** Installing app *build* in the Singularity container can take about 10 to 15 minutes.

[13]LIKWID is available from the following URL: https://github.com/RRZE-HPC/likwid/releases/tag/v5.0.1.

[14]Offsite is available from the following URL: https://doi.org/10.5281/zenodo.4283107.

[15]YASK is available from the following URL: https://github.com/intel/yask/tree/045b582.

- **How much time is needed to complete experiments (approximately)?:** For YaskSite runs (see Fig. 4–7), it takes about half an hour. For tuning runs with Offsite (see Fig. 8) it can take up to half a day.
- **Publicly available:** Yes.
- **Code License:** GNU Affero General Public License v3.0
- **Archived:** https://doi.org/10.5281/zenodo.4415588.

C. Description

1) Distribution: The artifact is publicly available[16]. It contains a Singularity container which handles the installation of all required tools and dependencies. Further, Singularity apps are provided to run the experiments shown in the paper. All required software components are automatically installed when executing the Singularity container. As their source codes are open-source, they are also separately retrievable[14,17].

2) Hardware Dependencies: YaskSite will work on most Intel architectures from the Sandy Bridge generation to modern Cascade Lake processors along with AMD Zen and Zen 2 architectures. To get comparable results with those provided in the paper, the Intel Cascade Lake (CLX) respectively AMD Rome (ROME) architecture are required. In particular, we used the two systems described in Table I.

The machine configurations used during our experiments are deposited in the artifact's folder *mc_state*. We recommend using these setting or similar settings. In particular, the CLX is configured with one ccNUMA domain per socket (Sub-NUMA Clustering off) and ROME with NPS1 mode.

3) Software Dependencies: All software dependencies have been resolved in the Singularity container.

D. Installation

Before installation, clone/download the artifact repository[16]. There you can find a README which explains the following steps in more detail. First, you need to download and install the Singularity engine using script *install_singularity.sh* provided in the artifact repository. Next, you can download the pre-build Singularity container[18]. Once the Singularity container is downloaded, the next step is to run app `build` in order to install YaskSite and Offsite:

```
singularity run --app build YS_CGO.sif
```

Executing this step at runtime is necessary, since YaskSite does machine specific configuration at build time.

E. Experiment Workflow

The artifact provides Singularity apps to reproduce the experimental results presented in the paper. The available apps can be listed using:

```
singularity run-help YS_CGO.sif
```

To reproduce for example the results in Fig. 4, Singularity app *Fig4* should be used. The following command will provide more information on the *Fig4* app, in particular its inputs and outputs.

```
singularity run-help --app Fig4 YS_CGO.sif
```

Please refer to the README for more details.

F. Evaluation & Expected Results

After installation the provided Singularity apps corresponding to different experiments (figures) have to be run. The apps output CSV files containing specific prediction and/or measurement results depending on the experiment. Similar results as in the paper are expected if the experiments are conducted on the systems mentioned in Section IX-C2 with corresponding settings. Runtime variations between different measurement runs were less than five percent.

G. Experiment Customization

In addition to the Singularity apps for reproducing the experiment results, own experiments can be run using the generic YaskSite and Offsite apps provided in the Singularity container.

[16]The artifact can be downloaded from the following URL: https://doi.org/10.5281/zenodo.4415588.

[17]YaskSite is available from the following URL: https://doi.org/10.5281/zenodo.4283028.

[18]The pre-build Singularity container is available from the following URL: https://doi.org/10.5281/zenodo.4415558

References

[1] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in High-Performance Computing Applications," *Proc. of the IEEE*, vol. 106, no. 11, pp. 2068–2083, Nov. 2018. https://doi.org/10.1109/JPROC.2018.2841200

[2] A. Tiwari and J. K. Hollingsworth, "Online Adaptive Code Generation and Tuning," in *Proc. 2011 IEEE Int. Parallel Distributed Processing Symp.*, ser. IPDPS '11. IEEE, May 2011, pp. 879–892. https://doi.org/10.1109/IPDPS.2011.86

[3] A. Rasch and S. Gorlatch, "ATF: A Generic Directive-Based Auto-Tuning Framework," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 5, p. e4423, Mar. 2019. https://doi.org/10.1002/cpe.4423

[4] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, Jan. 2001. https://doi.org/10.1016/S0167-8191(00)00087-9

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology," in *Proc. 11th Int. Conf. on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, Jul. 1997, pp. 340–347. https://doi.org/10.1145/263580.263662

[6] C. Yount, J. Tobin, A. Breuer, and A. Duran, "YASK – Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning," in *2016 6th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC. IEEE, Nov. 2016, pp. 30–39. https://doi.org/10.1109/WOLFHPC.2016.08

[7] P. Pfaffe, T. Grosser, and M. Tillmann, "Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping," in *Proc. ACM Int. Conf. Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 354–366. https://doi.org/10.1145/3330345.3330377

[8] J. Seiferth, C. Alappat, M. Korch, and T. Rauber, "Applicability of the ECM Performance Model to Explicit ODE Methods on Current Multi-core Processors," in *High Performance Computing: Proc. 33rd Int. Conf., ISC High Performance 2018*, ser. ISC '18. Berlin, Heidelberg: Springer, Jun. 2018, pp. 163–183. https://doi.org/10.1007/978-3-319-92040-5_9

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. https://doi.org/10.1145/1498765.1498785

[10] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model," in *Proc. 29th ACM Int. Conf. on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 207–216. https://doi.org/10.1145/2751205.2751240

[11] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels," in *Tools for High Performance Computing 2016*. Cham: Springer, Oct. 2017, pp. 1–22. https://doi.org/10.1007/978-3-319-56702-0_1

[12] G. Rivera and Chau-Wen Tseng, "Tiling Optimizations for 3D Scientific Computations," in *SC '00: Proc. 2000 ACM/IEEE Conf. on Supercomputing*, Nov 2000, pp. 32–32. https://doi.org/10.1109/SC.2000.10015

[13] M. Frigo and V. Strumpen, "The Memory Behavior of Cache Oblivious Stencil Computations," *The Journal of Supercomputing*, vol. 39, no. 2, pp. 93–112, Feb. 2007. https://doi.org/10.1007/s11227-007-0111-y

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, Jun. 2008. https://doi.org/10.1145/1379022.1375595

[15] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization," in *2009 33rd Annual IEEE Int. Computer Software and Applications Conf.*, vol. 1, July 2009, pp. 579–586. https://doi.org/10.1109/COMPSAC.2009.82

[16] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures," in *2011 IEEE Int. Parallel Distributed Processing Symp.*, May 2011, pp. 676–687. https://doi.org/10.1109/IPDPS.2011.70

[17] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015. https://doi.org/10.1137/140991133

[18] L. Peng, R. Seymour, K. ichi Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong, "High-order Stencil Computations on Multicore Clusters," in *2009 IEEE Int. Symp. on Parallel Distributed Processing*, May 2009, pp. 1–11. https://doi.org/10.1109/IPDPS.2009.5161011

[19] C. Yount, "Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation," in *2015 IEEE 17th Int. Conf. on High Performance Computing and Communications, 2015 IEEE 7th Int. Symp. on Cyberspace Safety and Security, and 2015 IEEE 12th Int. Conf. on Embedded Software and Systems*. IEEE, Aug 2015, pp. 865–870. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27

[20] T. Malas, G. Hager, H. Ltaief, and D. Keyes, "Multidimensional Intratile Parallelization for Memory-Starved Stencil Computations," *ACM Transactions on Parallel Computing*, vol. 4, no. 3, Dec. 2017. https://doi.org/10.1145/3155290

[21] P. van der Houwen and B. Sommeijer, "Parallel Iteration of High-order Runge-Kutta Methods with Stepsize Control," *Journal of Computational and Applied Mathematics*, vol. 29, no. 1, pp. 111–127, Jan. 1990. https://doi.org/10.1016/0377-0427(90)90200-J

[22] J. Seiferth, M. Korch, and T. Rauber, "Offsite Autotuning Approach," in *High Performance Computing: Proc. 35rd Int. Conf., ISC High Performance 2020*, ser. ISC '20. Cham: Springer, Jun. 2020, pp. 370–390. https://doi.org/10.1007/978-3-030-50743-5_19

[23] C. L. Alappat, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein, "Understanding HPC Benchmark Performance on Intel Broadwell and Cascade Lake Processors," in *High Performance Computing: Proc. 35rd Int. Conf., ISC High Performance 2020*, ser. ISC '20. Cham: Springer, Jun. 2020, pp. 412–433. https://doi.org/10.1007/978-3-030-50743-5_21

[24] J. Treibig, G. Hager, and G. Wellein, "likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes," in *Tools for High Performance Computing 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 27–36. https://doi.org/10.1007/978-3-642-31476-6_3

[25] Advanced Micro Devices, "Socket SP3 Platform NUMA Topology for AMD Family 17h Models 30h–3Fh," Retrieved April 21, 2020 from https://developer.amd.com/wp-content/resources/56338_1.00_pub.pdf, Oct 2019.

[26] J. Hofmann, C. Alappat, G. Hager, D. Fey, and G. Wellein, "Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors," *Supercomputing Frontiers and Innovations*, vol. 7, no. 2, pp. 54–78, 2020. https://doi.org/10.14529/jsfi200204

[27] I. Hirsh and G. S., "Intel Architecture Code Analysis," Retrieved April 21, 2020 from https://software.intel.com/en-us/articles/intel-architecture-code-analyzer, Apr 2020.

[28] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein, "Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, ser. PMBS '18. IEEE, Nov. 2018, pp. 121–131. https://doi.org/10.1109/PMBS.2018.8641578

[29] J. Laukemann, J. Hammer, G. Hager, and G. Wellein, "Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, ser. PMBS '19. IEEE, Nov. 2019, pp. 1–6. https://doi.org/10.1109/PMBS49563.2019.00006

[30] J. Hornich, J. Hammer, G. Hager, T. Gruber, and G. Wellein, "Collecting and Presenting Reproducible Intranode Stencil Performance: INSPECT," *Supercomputing Frontiers and Innovations*, vol. 6, no. 3, pp. 4–25, 2019. https://doi.org/10.14529/jsfi190301

[31] J. Hofmann, G. Hager, and D. Fey, "On the Accuracy and Usefulness of Analytic Energy Models for Contemporary Multicore Processors," in *High Performance Computing: Proc. 33rd Int. Conf., ISC High Performance 2018*, ser. ISC '18. Berlin, Heidelberg: Springer, Jun. 2018, pp. 22–43. https://doi.org/10.1007/978-3-319-92040-5_2

[32] M. Scherg, J. Seiferth, M. Korch, and T. Rauber, "Performance Prediction of Explicit ODE Methods on Multi-Core Cluster Systems," in *Proc. 2019 ACM/SPEC Int. Conf. on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019, pp. 139–150. https://doi.org/10.1145/3297663.3310306

[33] F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman, "Architecture and Performance of Devito, a System for Automated Stencil Computation," *ACM Transactions on Mathematical Software*, vol. 46, no. 1, Apr. 2020. https://doi.org/10.1145/3374916

[34] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication," *ACM Transactions on Parallel Computing*, vol. 7, no. 3, Jun. 2020. https://doi.org/10.1145/3399732

[35] M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein, "Chip-Level and Multi-Node Analysis of Energy-Optimized Lattice Boltzmann CFD Simulations," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 2295—2315, May 2016. https://doi.org/10.1002/cpe.3489

[36] F. Cremonesi, G. Hager, G. Wellein, and F. Schürmann, "Analytic performance modeling and analysis of detailed neuron simulations," *The International Journal of High Performance Computing Applications*, 2020. https://doi.org/10.1177/1094342020912528