

# Supercomputing Frontiers and Innovations

2019, Vol. 6, No. 3

## Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

## Editorial Board

### Editors-in-Chief

- **Jack Dongarra**, University of Tennessee, Knoxville, USA
- **Vladimir Voevodin**, Moscow State University, Russia

### Editorial Director

- **Leonid Sokolinsky**, South Ural State University, Chelyabinsk, Russia

### Associate Editors

- **Pete Beckman**, Argonne National Laboratory, USA
- **Arndt Bode**, Leibniz Supercomputing Centre, Germany
- **Boris Chetverushkin**, Keldysh Institute of Applied Mathematics, RAS, Russia
- **Alok Choudhary**, Northwestern University, Evanston, USA

- **Alexei Khokhlov**, Moscow State University, Russia
- **Thomas Lippert**, Jülich Supercomputing Center, Germany
- **Satoshi Matsuoka**, Tokyo Institute of Technology, Japan
- **Mark Parsons**, EPCC, United Kingdom
- **Thomas Sterling**, CREST, Indiana University, USA
- **Mateo Valero**, Barcelona Supercomputing Center, Spain

## Subject Area Editors

- **Artur Andrzejak**, Heidelberg University, Germany
- **Rosa M. Badia**, Barcelona Supercomputing Center, Spain
- **Franck Cappello**, Argonne National Laboratory, USA
- **Barbara Chapman**, University of Houston, USA
- **Yuefan Deng**, Stony Brook University, USA
- **Ian Foster**, Argonne National Laboratory and University of Chicago, USA
- **Geoffrey Fox**, Indiana University, USA
- **Victor Gergel**, University of Nizhni Novgorod, Russia
- **William Gropp**, University of Illinois at Urbana-Champaign, USA
- **Erik Hagersten**, Uppsala University, Sweden
- **Michael Heroux**, Sandia National Laboratories, USA
- **Torsten Hoefer**, Swiss Federal Institute of Technology, Switzerland
- **Yutaka Ishikawa**, AICS RIKEN, Japan
- **David Keyes**, King Abdullah University of Science and Technology, Saudi Arabia
- **William Kramer**, University of Illinois at Urbana-Champaign, USA
- **Jesus Labarta**, Barcelona Supercomputing Center, Spain
- **Alexey Lastovetsky**, University College Dublin, Ireland
- **Yutong Lu**, National University of Defense Technology, China
- **Bob Lucas**, University of Southern California, USA
- **Thomas Ludwig**, German Climate Computing Center, Germany
- **Daniel Mallmann**, Jülich Supercomputing Centre, Germany
- **Bernd Mohr**, Jülich Supercomputing Centre, Germany
- **Onur Mutlu**, Carnegie Mellon University, USA
- **Wolfgang Nagel**, TU Dresden ZIH, Germany
- **Alexander Nemukhin**, Moscow State University, Russia
- **Edward Seidel**, National Center for Supercomputing Applications, USA
- **John Shalf**, Lawrence Berkeley National Laboratory, USA
- **Rick Stevens**, Argonne National Laboratory, USA
- **Vladimir Sulimov**, Moscow State University, Russia
- **William Tang**, Princeton University, USA
- **Michela Taufer**, University of Delaware, USA
- **Andrei Tchernykh**, CICESE Research Center, Mexico
- **Alexander Tikhonravov**, Moscow State University, Russia
- **Eugene Tyrtynnikov**, Institute of Numerical Mathematics, RAS, Russia
- **Roman Wyrzykowski**, Czestochowa University of Technology, Poland
- **Mikhail Yakobovskiy**, Keldysh Institute of Applied Mathematics, RAS, Russia

## Technical Editors

- **Yana Kraeva**, South Ural State University, Chelyabinsk, Russia
- **Mikhail Zymbler**, South Ural State University, Chelyabinsk, Russia
- **Dmitry Nikitenko**, Moscow State University, Moscow, Russia

## Contents

<b>Collecting and Presenting Reproducible Intranode Stencil Performance: INSPECT</b> J. Hornich, J. Hammer, G. Hager, T. Gruber, G. Wellein .....	4
<b>Supercomputer Docking</b> A.V. Sulimov, D.C. Kutow, Vl.B. Sulimov .....	26
<b>Automatic Port to OpenACC/OpenMP for Physical Parameterization in Climate and Weather Code Using the CLAW Compiler</b> V. Clement, P. Mart, X. Lapillonne, O. Fuhrer, W. Sawyer .....	51
<b>Optimizing Deep Learning RNN Topologies on Intel Architecture</b> K. Banerjee, E. Georganas, D.D. Kalamkar, B. Ziv, E. Segal, C. Anderson, A. Heinecke .....	64
<b>A Skewed Multi-banked Cache for Many-core Vector Processors</b> H. Takayashiki, M. Sato, K. Komatsu, H. Kobayashi .....	86



This issue is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

# Collecting and Presenting Reproducible Intranode Stencil Performance: INSPECT

**Julian Hornich<sup>1</sup>, Julian Hammer<sup>1</sup>, Georg Hager<sup>1</sup>, Thomas Gruber<sup>1</sup>, Gerhard Wellein<sup>2</sup>**

© The Authors 2019. This paper is published with open access at SuperFri.org

Stencil algorithms have been receiving considerable interest in HPC research for decades. The techniques used to approach multi-core stencil performance modeling and engineering span basic runtime measurements, elaborate performance models, detailed hardware counter analysis, and thorough scaling behavior evaluation. Due to the plurality of approaches and stencil patterns, we set out to develop a generalizable methodology for reproducible measurements accompanied by state-of-the-art performance models. Our open-source toolchain and collected results are publicly available in the “Intranode Stencil Performance Evaluation Collection” (INSPECT).

We present the underlying methods, models and tools involved in gathering and documenting the performance behavior of a collection of typical stencil patterns across multiple architectures and hardware configuration options. Our aim is to endow performance-aware application developers with reproducible baseline performance data and validated models to initiate a well-defined process of performance assessment and optimization. All data is available for inspection: source code, produced assembly, performance measurements, hardware performance counter data, single-core and multicore Roofline and ECM (execution-cache-memory) performance models, and machine properties. Deviations between measured performance and performance models become immediately evident and can be investigated. We also give hints as to how INSPECT can be used in practice for custom code analysis.

*Keywords:* *performance modeling, performance analysis, stencils, single-node, multi-core, ECM, Roofline, memory hierarchy, cache effects.*

## Introduction

Stencils are relative data access and computation patterns that emerge from the discretization of differential operators on regular grids. Stencils appear in many fields, from image processing, fluid dynamics, material sciences to mechanical engineering, and are typically embedded in loop nests that are at least as deep as the number of dimensions in the original continuous problem. Despite their apparent simplicity, stencil algorithms show a rich set of performance patterns and allow for various optimizations in terms of data access and work reduction. For instance, the performance of most simple stencil algorithms (such as the 3D 7-point constant-coefficient variant encountered with a simple finite-difference discretization of the Laplace operator on a regular Cartesian grid) is limited by the memory bandwidth for in-memory working sets on multicore CPUs. Spatial blocking can reduce the code balance to a theoretical minimum but will not decouple from memory bandwidth. Temporal blocking can finally render the implementation cache or core bound with significant performance gains, but there are many different approaches and the number of parameters is significant [32]. Moreover, even recent publications often fail to assess performance baselines correctly, rendering all reported speedups meaningless.

We set out to compile an extensible collection of stencil runtime performance characteristics from a variety of architectures and backed up by analytic performance models and performance

---

<sup>1</sup>Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

<sup>2</sup>Department for Computer Science at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

counter analysis. The stencil update iteration is embedded in a Jacobi algorithm without advanced optimizations. All resulting information is available in a public online collection, organized based on an abstract classification scheme, and can be easily reproduced with the presented information and available open-source tool chain. The collection is also enriched with specific comments on the performance behavior and model predictions. It can be browsed at [24].

Our work makes the following contributions:

- simple classification scheme of stencil characteristics, based on the underlying numerical problem, and defining the architecture dependent performance behavior (see Sec. 1);
- support for automatic analytic performance model generation for the AMD Zen microarchitecture (see Sec. 4.2);
- first-of-its-kind collection, presentation and method to match the measured performance data with automatically generated single- and multicore performance models in order to gain insight into relevant performance bottlenecks and uncover compiler deficiencies and to guide performance optimization strategies (see Secs. 3, 4 and 5);
- automatic extraction of the Phenomenological ECM model from hardware performance counters (see Sec. 2.5.3), based on ideas from [32];
- public website on which the gathered data, performance models, and reproducibility information are presented in a clear and structured way (see [24]);
- built-in reproducibility by transparently making all necessary runtime information and system configurations available to developers—including the exact commands to execute for reproduction (see Sec. 4).

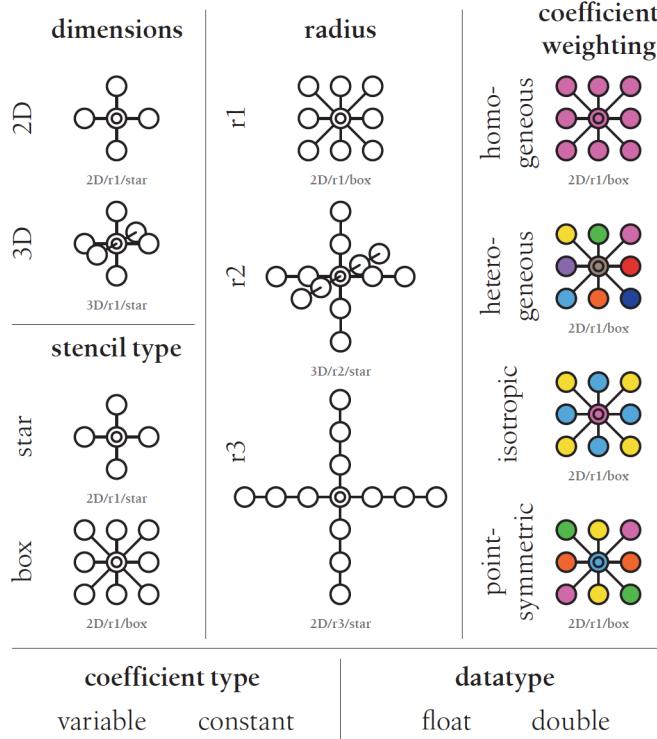
This paper is organized as follows: Section 1 introduces our stencil classification scheme. In Section 2 we briefly describe the computer architectural features of the benchmark systems, the analysis tools, and the performance models we employ to set up the stencil performance collection. Section 3 details on our automated workflow and includes a description of the structure and origin of the data presented on the INSPECT website. Section 4 uses a few distinct stencil examples to showcase the data presentation and possible insights gained. Section 6 gives related work. The article concludes with a summary of the main results and an outlook onto the future heading of this work.

## 1. Stencil Classification

In order to span a space of possible stencils we use a classification based on the following properties:

- *dimensions*: dimensionality of the stencil, typically 3D or 2D;
- *radius*: longest relative offset from the center point in any dimension, usually  $r = 1$ ,  $r = 2$ , or  $r = 3$ ;
- *coefficient weighting*: how coefficients are applied, e.g. homogeneous, heterogeneous, isotropic, or point-symmetric;
- *stencil type*: general shape of stencil, e.g., star or box;
- *coefficient type*: constant throughout grid or variable at each grid point;
- *data type*: numeric type of grid elements, e.g., double or float.

These properties may have a large performance impact depending on the details of the underlying CPU architecture and its features and configuration, making runtime predictions difficult. Much of the complexity lies in the cache hierarchy, where data transfers may be handled before they reach main memory, and behavior depends on spatial and temporal locality. On



**Figure 1.** Classification of stencils used by INSPECT. These properties do not capture all aspects, but are representative for a large number of discretizations. Colors represent applied coefficients

the other hand, in-core bottlenecks such as pipeline latencies and throughput limits, may also play a decisive role, especially with more complicated stencils. A visual overview of the stencil classification is given in Fig. 1. Isotropic coefficient weighting deserves special attention: all nodes with the same distance to the origin share the same coefficient; different distances have distinct coefficients.

With the given set of classification properties this leads to at least 192 relevant combinations. We have not yet gathered data for all possible combinations and architectures available to us, but a representative set is already available.

## 2. Background & Methodology

To support our methodology, we use STEMPEL [10] for code generation based on the classification shown above, and Kerncraft [14] for the generation of Roofline and Execution-Cache-Memory (ECM) models based on micro-architectural and memory hierarchy features as well as benchmarking of single- and multi-core scenarios.

We will briefly introduce STEMPEL, Kerncraft and their underlying models ECM and Roofline, as well as hardware features which have a significant impact on similar codes.

### 2.1. STEMPEL

STEMPEL is used to generate stencil codes from the parameters mentioned in Sec. 1 (dimensions, radius, coefficient weighting, stencil type, coefficient type and data type). The resulting kernel code is used as input for Kerncraft, but STEMPEL also supports generation of benchmarkable code which can be compiled and executed stand-alone. The generated code may also include OpenMP multithreading support or spatial blocking. The latter is used to investigate

blocking behavior for INSPECT. For accurate extraction of performance data, the code is additionally instrumented with LIKWID markers to be used with the `likwid-perfctr` tool.

Note that STEMPER produces plain C code at the time of writing, but hardware-aware programming with SIMD intrinsics (see, e.g., [28]) sometimes promises better performance. Using SIMD intrinsics for SSE, AVX, and AVX512 instruction set extensions would be possible, but analysis of such code is currently not possible with Kerncraft (although planned for future work). Moreover, since the compiler-generated assembly code is readily available in INSPECT, failure of the compiler to produce efficient SIMD instructions will immediately be apparent. Note that such situations most often occur when the stencil code is embedded in an environment that obscures the compiler’s view on the relevant details, such as nonoverlapping arrays, etc. Under these conditions, it is hard to prove that the required code transformations for effective vectorization are allowed. Since INSPECT invokes the compiler on a simple source code in a very clean setting, vectorization is usually not a problem for the stencil structures investigated here.

## 2.2. ECM & Roofline Model

The Execution-Cache-Memory (ECM) [38] and Roofline [40] models are resource-centered performance models that assume certain hardware limitations (such as data transfer bandwidths, instruction throughput limits, instruction latencies, etc.) and try to map the application to a simplified version of the hardware in order to expose the relevant bottleneck(s). This analysis depends on the dataset size and dimensions, as well as the computational effort during each iteration. Both models generally neglect data access latencies although they can be added as “penalties”. Latency predictions would require other models and are usually not of relevance for stencil code performance. In some cases, latency penalties need to be considered for “perfect” predictions [18], but this correction is usually small and is neglected in this work.

The compute performance bottleneck is analyzed based on the loop body’s maximum in-core performance. Assuming that all load operations will hit the L1 cache, one can estimate the optimistic runtime the loop body in cycles. The necessary information has been published by Intel, Agner Fog [6], uops.info [1], or through the Intel Architecture Code Analyzer (IACA) [15]. Although none of those sources are complete, they are good enough for a well-informed estimation. The resulting inverse throughput of cycles per cacheline (lower is faster) exposes the bottleneck, for both ECM and Roofline. Cachelines are considered as the basic work unit, since it is also the basic unit of caches. E.g., for a double precision code with 8 Byte per element, on a machine with 64 Byte cachelines, there are eight iterations per cacheline. For Roofline most publications use performance (higher is faster) as the baseline metric, but both units can be converted into one another trivially:

$$\text{clock}/\text{performance} \times \text{work} = \text{inverse throughput}$$

$$\left[ \frac{\text{cycle}}{\text{second}} \right] / \left[ \frac{\text{FLOP}}{\text{second}} \right] \times \left[ \frac{\text{FLOP}}{\text{iteration}} \right] \times \left[ \frac{\text{iteration}}{\text{cacheline}} \right] = \left[ \frac{\text{cycle}}{\text{cacheline}} \right]$$

Memory and cache bottlenecks require a prediction of which data access will be served by which memory hierarchy level. This is either done using a cache simulator (e.g. `pycachesim` [12]) or the analytical layer-condition model [11, 14]. The result from this prediction are the expected traffic volumes between the levels of the memory hierarchy. The Roofline model then combines the data volume per cacheline (e.g., eight iterations with double precision) for each hierarchy level with previously measured bandwidths for the same level and core count,

**Table 1.** Host configuration used for INSPECT. See [25] for all details

Host	HSW	BDW	SKX	ZEN
CPU model	Intel Xeon E5-2695v3	Intel Xeon E5-2697v4	Intel Xeon Gold 6148	AMD EPYC 7451
base clock (fixed)	2.3 GHz	2.3 GHz	2.4 GHz	2.3 GHz
uncore clock (fixed)	2.3 GHz	2.3 GHz	2.4 GHz	n/a
turbo mode	disabled	disabled	disabled	disabled
cores per socket	14	18	20	24
cores per NUMA domain	7	9	10	6
cluster-on-die (CoD) / sub-NUMA-clustering (SNC)	CoD enabled  Haswell see Fig. 2a	CoD enabled  Broadwell see Fig. 2a	SNC enabled  Skylake X see Fig. 2b	n/a  Zen see Fig. 3
INSPECT machine name	BroadwellEP_E5-2697_CoD	BroadwellEP_E5-2697_CoD	SkylakeSP_Gold-6148_SNC	Zen_EPYC-7451

and selects the slowest as the bottleneck. The ECM model combines all inter-cache transfers with theoretical bandwidths from documentation, and volumes between memory and last level cache with a measured full-socket bandwidth. These inverse throughputs are either combined using summation if no overlapping is assumed, maximization for full overlapping, or any more complicated function for intermediate situations.

For Intel processors, assuming that no overlap between any load, store, inter-cache and memory transfers happens has proven to be the best-fitting model assumption. This might change in future microarchitectures and does not hold for other vendors.

For the AMD Zen microarchitecture, in-core computations and all inter-cache and register transfers overlap down to the L2 cache. Transfers between L2, L3 and main memory serialize [16].

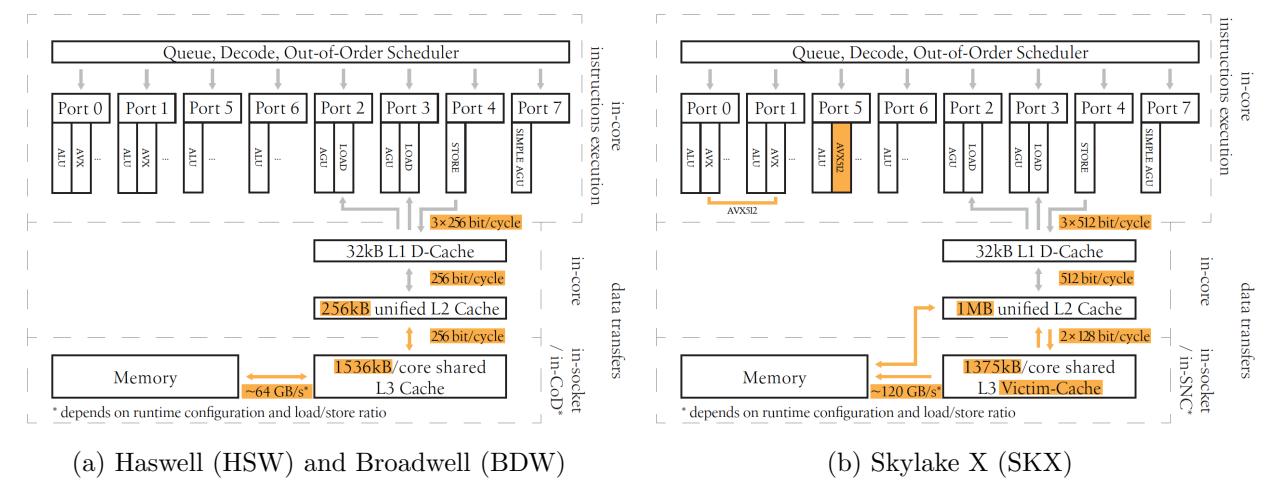
Another approach is to measure transfers using hardware provided performance counters and base the ECM model on these empirical volumes and predict the runtime using the non-overlapping assumption. This is referred to as the *Phenomenological ECM model*, also discussed in Sec. 2.5.3.

The model parameters used by the models are shown in Fig. 2 and Tab. 1. The corresponding machine files can be found on the INSPECT webpage. For the ECM model parameters, except for the measured memory bandwidth highlighted by the preceding tilde (~), all throughputs are published in vendor documentation. The throughputs of execution ports, scheduler and decoder are not shown here, but most may be found in official documentation, public resources, or can be benchmarked [13]. The overall instruction-level parallelism capability is represented with the different ports.

While in this work the Roofline model is always presented with a single (reciprocal) throughput (TP), the ECM model is produced from architecture-dependent combinations of in-core computation TP  $T_{\text{comp}}$ , load/store TP  $T_{\text{RegL1}}$ , inter-L1/L2 transfer TP  $T_{\text{L1L2}}$ , inter-L2/L3 transfer TP  $T_{\text{L2L3}}$  and main memory transfer TP  $T_{\text{L3MEM}}$ ), with the unit of cycles per cacheline.

### 2.3. Intel Microarchitectures

The Intel microarchitectures Haswell (HSW) and Broadwell (BDW) have no differences in regard to our modeling and performance analysis. Figure 2a shows their architectural diagram. Both architectures have seven execution ports; most important are the two AVX2 fused-multiply-add (FMA) ports, two load ports able to handle 256-bit per cycle and one 256-bit store port. AVX and AVX2 instructions can make use of sixteen 256-bit YMM registers. On the memory side, there is a linear inclusive cache hierarchy. Due to the double ring interconnect between individual



**Figure 2.** Simplified block diagrams for Intel Haswell, Broadwell and Skylake X, including the execution ports and cache hierarchy. Differences have been highlighted. The Skylake architecture (without X) has no AVX512 support

HSW and BDW cores and separate memory controllers residing on each ring, a cluster-on-die mode can be enabled to allow NUMA separation of the two rings. With cluster-on-die mode, the last-level cache (i.e., L3) is split and only used by a core on the same ring, and a slightly higher memory bandwidth can be attained. Results for HSW and BDW are presented with cluster-on-die (CoD) mode enabled.

The Skylake X (SKX) microarchitecture is shown in Fig. 2b. It supports AVX512, which boosts load, store and FMA ports from 256-bit to 512-bit width. To allow two AVX512 FMA instructions to be executed in parallel (i.e., combined throughput of 0.5 cycles) an AVX512 pipeline was added to Port 5 and the existing 256-bit pipelines at Ports 0 and 1 may be used in lockstep to reach  $2 \times 512$ -bit width. There are 32 512-bit YMM registers and the number of 256-bit registers was also doubled to 32. The SKX microarchitecture has a non-inclusive last level victim cache, which may cache evicted cachelines from L2 and is used to write back to but not to load from memory. All data coming from memory is loaded directly into the L2 cache of the requesting core. Unmodified cachelines may also be dropped from L2. The criteria which decide if a cacheline is evicted from L2 to the victim cache or not have not been disclosed. For our models we assume that all evicts will be passed on to the last-level cache and—if changed—stored from there into memory. Sub-NUMA-clustering (SNC) is similar to CoD, which was also enabled during our measurements.

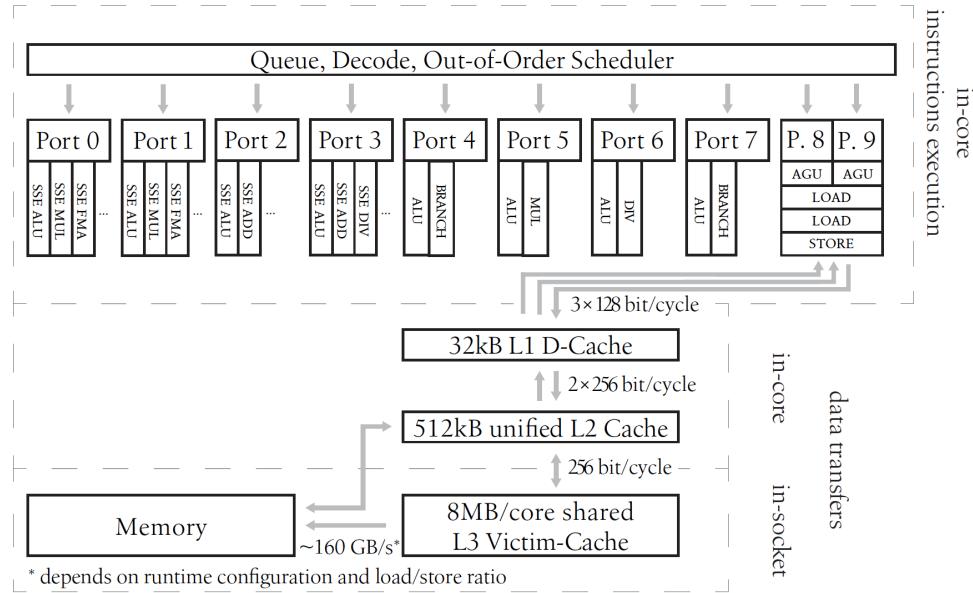
The changed cache structure of the Skylake microarchitecture, with its undocumented decision heuristic for L3 cache usage and unavailable hardware counters for some of the inter-cache and memory data paths, still poses a problem. As we will see later, assuming the traditional linear inclusive cache hierarchy often yields reasonable results, but is still under investigation.

In the architecture diagrams, two-way arrows represent half-duplex capabilities, and two individual arrows mean full-duplex capabilities. The factor along with the bandwidth is meant to emphasize the half- and full-duplex behavior (e.g., between L2 and L3 on Skylake X 128 bits per cycle may be transferred both ways concurrently). This information and more is used to construct a machine file for Kerncraft and will be explained in Sec. 2.5.1.

The specific systems that are used for INSPECT are documented at [25]. A summary of the relevant configuration details, in addition to the micro architectural details in Fig. 2, is given in Tab. 1.

## 2.4. AMD Zen Microarchitecture

The AMD Zen microarchitecture has ten ports, the first four of which (0, 1, 2 and 3) support 128-bit wide floating-point SSE instructions (see Fig. 3). Each execution unit, except for divide, is present on two ports, e.g., FMA and MUL on 0 and 1, and ADD on 2 and 3. The decoder stage supports AVX instructions by utilizing two SSE ports simultaneously (similar to AVX512 on Port 0 and 1 with Skylake X). Ports 4 through 7 handle integer and control flow instructions.



**Figure 3.** Simplified block diagram for AMD Zen microarchitecture, including execution ports and cache hierarchy

Ports 8 and 9 each have their own address generation unit (AGU) and can utilize the two shared load ports and the single shared store port. The store and load ports each operate on up to 128 bits and can issue one load/store to the 32 kB first-level (L1) cache. Thus either two loads or one load and one store can be executed per cycle at maximum. The 512 kB inclusive L2 cache is connected to the L1 cache with 256-bit per cycle full-duplex (i.e., a 64 B cacheline takes two cycles to transfer, and loads and stores are done in parallel). Between L2 and the last-level cache (L3), 256 bits can be transferred per cycle, but only half-duplex (i.e., either load or store). The exact heuristics of the victim cache are not publicly available. In addition to that, support for hardware performance counters is much more limited, which does not allow us to inspect many of the transfers between memory levels.

The maximum main memory bandwidth achievable is close to 160 GB/s, 30% higher than what we were able to measure on Skylake X. The specific AMD CPU used here has 24 cores, split over four NUMA domains of 6 cores each.

## 2.5. Kerncraft

Kerncraft brings together static analysis of the kernel code with microarchitecture data and execution models into a coherent performance prediction model, based on the Roofline and ECM models. It also allows benchmarking of the kernel codes with single and multiple cores (using OpenMP) and collection of hardware performance counter data during execution.

### 2.5.1. Machine model

The specific machine model for each microarchitecture is described in the *machine files*, which are provided either with Kerncraft or can be generated semi-automatically using `likwid_bench_auto`—a tool distributed with Kerncraft. All machine models mentioned in this paper are provided with Kerncraft in the examples directory. When using the semi-automatic generation, it must be executed on such a machine and the resulting file needs to be completed manually from vendor documentation, model assumptions or from existing—similar—architecture files.

Machine model files contain detailed information on the architecture and memory hierarchy, as well as benchmark results, necessary to construct the Roofline and ECM model. In particular STREAM [33] benchmark results, cache sizes and parameters, NUMA topology, base clock and architecture specific compiler arguments make up the majority of the description. Some of it can be collected automatically, some needs to be provided manually.

The INSPECT website presents a breakdown of the architecture information in the machine files [25]. There may be known issues, which are also documented on INSPECT. E.g., Haswell’s L1-L2 bandwidth is theoretically 64 Byte per cycle, but benchmarks show that the achievable bandwidth may be as low as 32 B/cy. Kerncraft assumes the optimistic 64 B/cy.

### 2.5.2. Model construction

The static analysis and model building is split in two parts: in-core execution and data analysis; both of which are done without execution of the kernel code and can be performed on any hardware.

The in-core analysis is done via the Intel Architecture Code Analyzer tool (IACA) [15] for Intel architectures and the Open Source Architecture Code Analyzer (OSACA) [29], which yields the number of cycles each execution port is occupied by the kernel’s assembly instructions ( $T_{\text{comp}}$  and  $T_{\text{RegL1}}$  for the ECM model). Kerncraft takes care of compilation, unrolling and vectorization in order to correctly interpret the IACA/OSACA result and relate it to high-level loop iterations found in the kernel source code.

The data analysis predicts inter-cache and memory transfer data volumes either using the analytical layer-condition model (LC) or the pycachesim [12] cache simulator. This yields  $T_{\text{L1L2}}$ ,  $T_{\text{L2L3}}$  and  $T_{\text{L3MEM}}$  for the ECM model. The LC model analysis is very fast and gives a closed form analytical model, but relies on an idealistic full-associative inclusive and least-recently-used cache hierarchy. The cache simulator can handle more realistic and complex cache configurations, such as associativity and non-inclusive cache hierarchies—at the cost of speed and without a closed form solution. Certain aspects of real hardware can not be simulated due to missing documentation, e.g., cache placement algorithm for last-level cache on current Intel microarchitectures.

For multi-core scaling, we use the memory latency penalty estimation as described by Hofmann [17].

### 2.5.3. Benchmark mode and phenomenological ECM

Benchmarking of any code can be tricky, this is the same for stencil codes. Kerncraft takes care of pinning and hardware performance counter monitoring with LIKWID [39], as well as ensuring a minimal runtime, checking machine configuration and derivation of relevant metrics

from the measurements. The underlying performance counters are defined in the machine model and based on validated metrics provided by LIKWID.

In addition, metrics based on measurement of the runtime, such as memory bandwidths and lattice updates per second, data transfer volumes can be measured accurately. From these Kerncraft can construct a *Phenomenological ECM model*. This phenomenological model is not based on the measured runtime or derived bandwidths, but uses inter-cache and memory data volumes, as well as counts of executed  $\mu$ ops per port. The overall prediction is then compiled in the same way as the analytical ECM prediction is compiled from vendor documented transfer rates, measured memory bandwidth and instruction throughput information.

The specific counters necessary have been compiled from Intel documentation and their correctness validated with microbenchmarks, where possible. This process is part of the ongoing LIKWID development. Kerncraft's machine models put the counter in relation to ECM model parameters, such as L1-L2 traffic or execution port utilization. To measure all necessary counters, multiple executions are unavoidable, because only a limited number of counter registers are available for use. On Intel's server microarchitectures, many performance counters are available, and a complete model can be assembled as presented in Fig. 5f. On AMD Zen, however, essential contributions, such as main memory traffic, can not be examined, and a complete phenomenological model may not be constructed.

### 3. Data Collection

In order to build a comprehensive single-node stencil performance database, preexisting open-source tools STEMPEL [10], Kerncraft [14] and LIKWID [39] have been combined in the “Intranode Stencil Performance Evaluation Collection” (INSPECT). For given stencil parameters all benchmark and automated performance modeling data for the present machine can be collected with a single (job) script. The data collection work flow can be seen in Alg. 1.

STEMPEL is used for generating the stencil source code that is to be fed into Kerncraft. Possible parameters are: dimension, radius, stencil type, coefficient weighting and type, as well as the data type. Examples of the produced stencil code by STEMPEL are shown in Fig. 4, 8 and 9. If a custom stencil is to be used, this step can be omitted.

The stencil code is then supplied to Kerncraft in order to do layer condition analysis and determine sensible ranges for grid sizes to be examined. Data ranges are chosen such that the last level cache 3D layer condition is violated and a steady state will be reached as long as the available main memory per NUMA domain is not exceeded (see  $1.5 \cdot LC_{L3,3D}$  in Alg. 1).

The next step is data collection. For single core grid scaling, Kerncraft is used to generate Roofline and ECM performance models with layer conditions and cache simulation, as well as benchmark and Phenomenological ECM data. Multi-core thread scaling is done for the largest previously calculated, memory bound grid size for all cores of one socket. Here Kerncraft is again used to generate Roofline, ECM and Benchmark data. In a last step spatial blocking is performed. Here STEMPEL is once more used to generate executable benchmark code with spatial blocking from the basic stencil code, generated before. This spatial blocking code is then instrumented with LIKWID to obtain the required benchmark data.

In a final step all data is collected, postprocessed and archived. The outputs are the data files that are needed for the visualization on the website. Those files can be pushed to the git repository to automatically include the inspected stencil on the INSPECT website.

**Algorithm 1** INSPECT data collection workflow

---

```

fix frequency
for dimension, radius, kind, coefficients, weighting, data type do
    stempel: generate stencil
    kerncraft: layer condition analysis
     $LC_{L3, 3D} \leftarrow$  compute L3 3D layer condition size
     $N_{L3, 3D'} \leftarrow 1.5 \cdot LC_{L3, 3D}$ 
    if  $N_{L3, 3D'} <$  available Memory per NUMA-domain then
        reduce grid_size until it fits in memory
    end if
    for  $n \leftarrow 10, N_{L3, 3D'}$ , do                                 $\triangleright$  single core grid scaling
        kerncraft: (LC, CS)  $\times$  (RooflineIACA, ECM)
        kerncraft: Benchmark
    end for
    for Threads  $\leftarrow 1, N_{threads}$  do                       $\triangleright$  thread scaling @  $N_{L3, 3D'}$ 
        kerncraft: (RooflineIACA, ECM, Benchmark)
    end for
    stempel: generate 3D spatial blocking code
    for  $n \leftarrow 10, N_{L3, 3D'}$ , do                                 $\triangleright$  single core grid scaling
        determine 'good' 3D blocking factors
        likwid-perfctr: Benchmark
    end for
    csv data + graphs + website                                 $\triangleright$  post processing
end for

```

---

For every stencil-machine configuration the website shows general stencil information, graphs of the measured and predicted performance, and step-by-step instructions for the replication of the shown data. The general stencil information contains stencil parameters, kernel source code, kernel assembly, and layer condition analysis, as well as IACA throughput analysis and information about the state of the machine and operating system the data was collected on. Performance prediction and benchmark data are shown in five different graph types:

- stacked ECM (with layer condition, cache simulation and phenomenological);
- Roofline performance (with layer condition and cache simulation);
- data transfers for single core grid scaling;
- full-socket thread scaling (one grid size by default, but possibly more);
- spatial blocking performance plots (3D-L3 cache blocking by default, but possibly more).

An example of the plots visible for each stencil configuration is shown in Fig. 5. The reproducibility information contains detailed steps on how to generate the stencil code with STEMPEL and all necessary commands to retrieve the data shown on the site.

Additionally, all data shown can be commented and validated with a traffic light system reflecting the quality of the plots. This allows to highlight problems or nonintuitive results of a specific stencil or hardware configuration, which could otherwise be mistaken for incorrect data.

## 4. Examples

Three different exemplary stencil configurations were selected from the INSPECT website: short-/long-ranged and star/boxed stencils, on three different machines. We will start with a very basic 7-point stencil on a Haswell Xeon E5-2695v3 machine, then continue with a long-ranged stencil on Skylake Xeon Gold 6148 and compare a boxed stencil on Broadwell Xeon E5-2697v4 and Skylake Xeon Gold 6148; we will conclude with the basic 7-point stencil on an AMD EPYC 7451 machine.

In addition to the presented architectures, the INSPECT website also contains analyses and measurements on Intel Sandy Bridge and Ivy Bridge architectures.

### 4.1. A Simple Short-ranged Stencil on Haswell (Intel Xeon E5-2695v3)

The first stencil configuration presented here is the simple 3D 7-point stencil on the well understood Haswell microarchitecture (Intel Xeon E5-2695v3 in CoD mode): 3D, radius 1, star stencil, constant and homogeneous coefficients with double precision floating-point accuracy. The stencil source code is shown in Fig. 4. Figure 5 displays all graphs presented on the INSPECT website, as well as the workflow from stencil generation with STEMPEL to data acquisition with Kerncraft, as already outlined by Alg. 1. The model prediction graphs show a stacked ECM prediction ( $T_{\text{ECM}} = \max(T_{\text{comp}}, \text{sum}(T_{\text{RegL1}}, T_{\text{L1L2}}, T_{\text{L2L3}}, T_{\text{L3MEM}}))$ ), together with the Roofline prediction and benchmark measurement data.

All the presented data and plots on this specific kernel, including commands to reproduce, system configuration used and very verbose information on the analysis (such as the IACA output) may be viewed at [23].

Figure 5a shows the ECM and Roofline prediction generated by Kerncraft, based on the layer condition prediction. Figure 5b is based on the cache simulation. The stacked colored regions represent the data transfers in the ECM model, where the upper boundary is equal to the ECM prediction including all data transfers ( $T_{\text{RegL1}} + T_{\text{L1L2}} + T_{\text{L2L3}} + T_{\text{L3MEM}}$ ). The red line represents the compute bottleneck ( $T_{\text{comp}}$ ), as predicted with IACA. The Roofline prediction is the thin blue line with circles. The black line with x's are the measurement results from running Kerncraft in benchmark mode.

The Roofline prediction is accurate when all layer conditions are violated and all stencil data is coming from main memory. Before that, its prediction is about 25% too optimistic. In the transition zone, there are a few points where the Roofline model is too pessimistic, because the

---

```

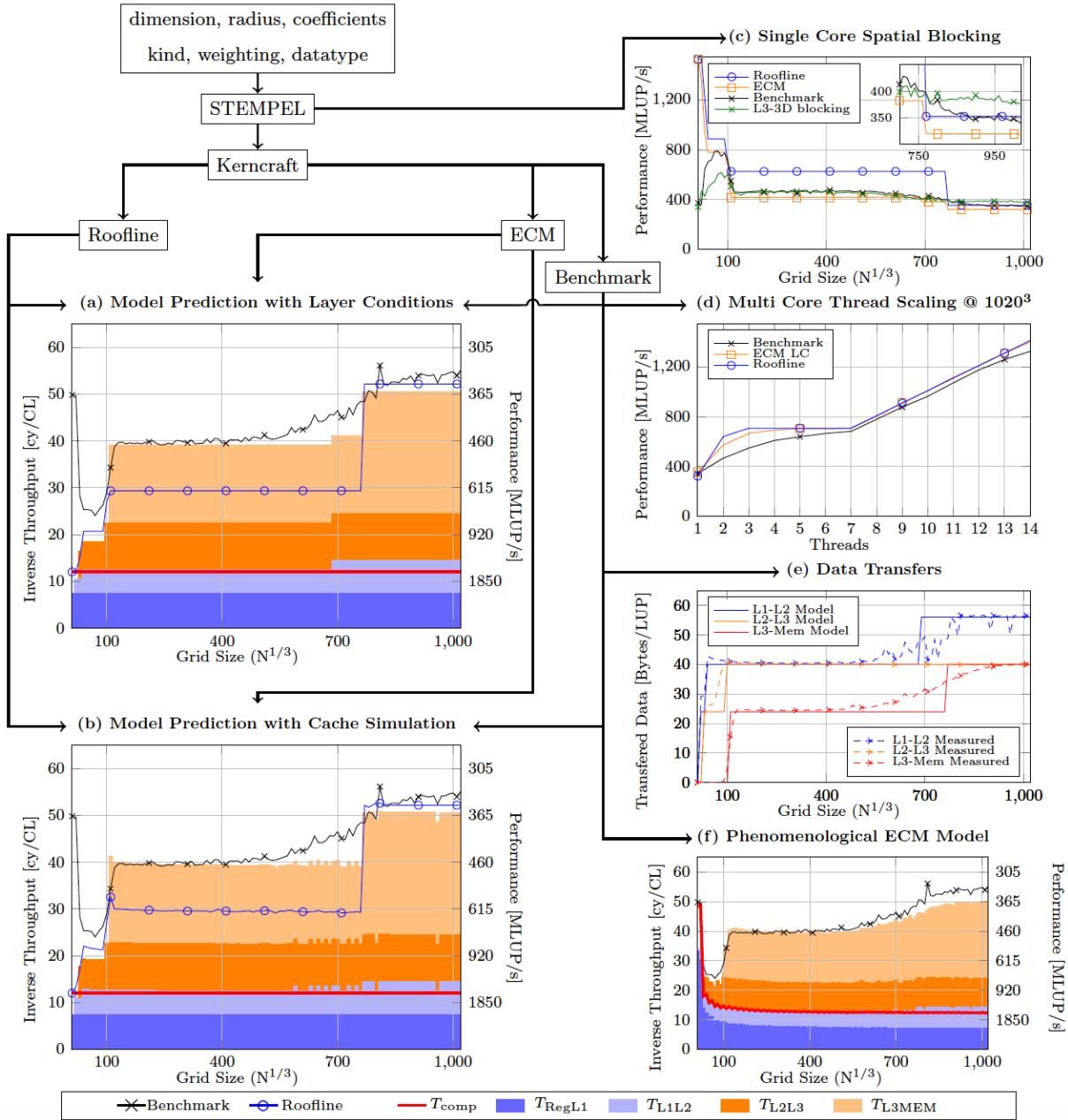
double a[M][N][P], b[M][N][P], c0;

for(long k = 1; k < M - 1; ++k) {
    for(long j = 1; j < N - 1; ++j) {
        for(long i = 1; i < P - 1; ++i) {
            b[k][j][i] = c0 * ( a[k][j][i]
                + a[k][j][i-1] + a[k][j][i+1]
                + a[k][j-1][i] + a[k][j+1][i]
                + a[k-1][j][i] + a[k+1][j][i] );
        }
    }
}

```

---

**Figure 4.** 3D stencil code with radius 1, constant homogeneous coefficients, star structure and double data type



**Figure 5.** Flow chart visualizing the connections between STEMPEL and Kerncraft and the resulting data. STEMPEL is used to generate the kernel and spatial blocking code. Kerncraft performs Roofline and ECM analysis on the kernel code for layer condition and cache simulation. Kerncraft’s benchmark mode provides the measurement data for grid scaling, data transfers and Phenomenological ECM, as well as multi core thread scaling. The shown data is for a 3D star stencil with radius 1 and constant, homogeneous coefficients with double precision floating-point numbers on a Haswell Xeon E5-2695v3 machine with Cluster on Die mode enabled. Underlying data, plots and additional model information may be viewed at [23] on the INSPECT website

mathematical layer condition is sharp but the performance shows a smooth transition because of the cache replacement policy. The ECM model results in a much more accurate prediction. All layer condition jumps ( $30^3$ : L1-3D,  $90^3$ : L2-3D,  $680^3$ : L1-2D,  $760^3$ : L3-3D) are clearly visible and correspond to the measured performance. The large deviation between models and measurement in the initial section ( $N < 100^3$ ) comes from loop overheads and large impacts of remainder loop iterations. While this is expected, it is neither modeled by ECM nor Roofline.

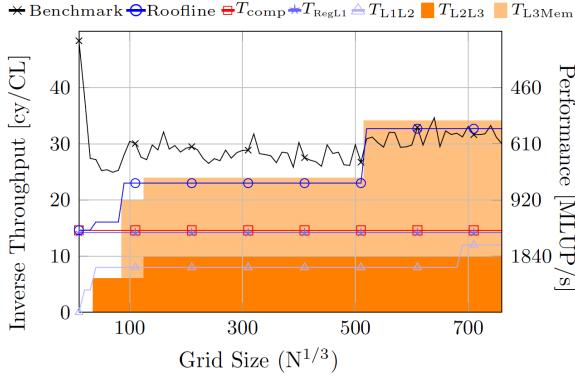
Comparing the cache simulator (Fig. 5b) with the layer conditions (Fig. 5a) shows that some peaks or dips in the measurement can be explained by using a more accurate cache model as provided by the cache simulator. It also allows for a smoother transition between broken layer conditions, but nonetheless fails at accurately predicting the transition behavior. Perfect tracking of those transitions, as seen in the benchmark measurements, would only be possible with precise knowledge of the underlying caching algorithms implemented in the different cache levels. Due to a lack of information from the CPU vendors a perfect LRU cache is assumed, as well as other idealized implementation details.

In the Phenomenological ECM graph, cf. Fig. 5f, the smooth transition between broken layer conditions can be tracked very well. Apart from the transitions zones, the individual contributions as modeled by the analytical ECM model (Fig. 5a and 5b) and derived from measurements in the Phenomenological ECM model match up very well. It also shows why the measured performance differs immensely from the predictions below  $100^3$ , which now shows as very high in-core execution time of the load instructions because of short scalar loops. The difference between measurement and model towards the right side of the graph hints that there are saturation effects in the memory interface which we do not yet understand fully.

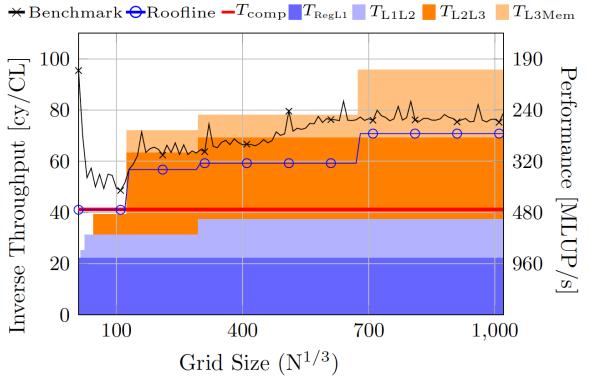
The data transfer volumes predicted by the layer conditions and their comparison with measured data volumes through hardware performance counters can be seen in Fig. 5e. The solid lines show the data transfer prediction between cache levels and main memory and the dashed lines are the measured data transfers. Between  $100^3$  and  $500^3$ , as well as after  $850^3$ , the predicted transfers at each level and the measured data fit perfectly and show the accuracy of this method. As layer condition break, the measured cache transfers show a smooth transition, until it realigns with the predicted data volume.

Figure 5c shows the impact of cache blocking for specific layer conditions. In this case, blocking was performed for the L3-3D layer condition, where only the middle loop (e.g.,  $j$ -loop in Fig. 4) is blocked to keep the relevant data at least in the L3 cache and reduce main memory traffic to a minimum. As intended, performance stays constant after the L3-3D layer condition is broken, with spatial cache blocking enabled (green line). This behavior can be predicted from Fig. 5a, where spatial blocking means to preserve the throughput of an earlier plateau while increasing the dataset size. Reasonable blocking factors are given by the range of the plateau (e.g., here the block dimensions should be  $N_{\text{block}}^{1/3} < 700$ ). Blocking for the next lower plateau (i.e.,  $N_{\text{block}}^{1/3} < 100$ ) may introduce too much overhead due to short loops. Another, more complicated option would be the use of temporal blocking, which is expected to yield about the same performance as  $N_{\text{block}}^{1/3} < 100$ , because stripping the top contribution from the stacked plot would bring the throughput to the same plateau.

Moving on from the single core to multi core scaling, Fig. 5d shows the in-socket thread scaling behavior at  $1020^3$ . Due to the Cluster-on-Die configuration of the machine, the performance flattens out at the end of the first NUMA-domain (7 cores). With the addition of the second NUMA-domain a linear increase can be seen, due to the linear addition of bandwidth from the



**Figure 6.** ECM and Roofline predictions with benchmark results for 3D 7-point star stencil on AMD ZEN



**Figure 7.** ECM and Roofline predictions for SKX, based on layer conditions, with benchmark results for 3D/3r/heterogeneous/star/constant/double stencil

added cores based on the compact scheduling scheme. Predictions of ECM and Roofline models fit very well in the second, linear part of the graph, and the ECM is also able to capture the phase before memory bandwidth saturation.

#### 4.2. A Simple Short-ranged Stencil on AMD Zen

In Fig. 6 we show an analysis on the AMD Zen microarchitecture, presenting results for the same kernel as in Fig. 4. These and additional results may be found at [22]. As described in Sec. 2.2, the AMD Zen architecture shows strong overlap in data transfers. The port execution model is based on the OSACA implementation [29], and the Kerncraft version used for this is based on the latest feature/OSACA branch. For data volume prediction we use the layer condition model, as for SKX.

The ECM prediction for the AMD Zen microarchitecture is based on the following model, which has fewer serializing terms:

$$T_{\text{ECM}} = \max(T_{\text{comp}}, T_{\text{RegL1}}, T_{\text{L1L2}}, \sum(T_{\text{L2L3}}, T_{\text{L3MEM}})).$$

This difference is also visible in Fig. 6, where the overlapping parallel terms ( $T_{\text{comp}}$ ,  $T_{\text{RegL1}}$  and  $T_{\text{L1L2}}$ ) are simple lines and the serial contribution terms ( $T_{\text{L2L3}}$  and  $T_{\text{L3MEM}}$ ) are stacked onto one another.

As with Skylake X, the benchmark follows the trend of the model qualitatively, but measurements yield better throughput with increased main memory traffic. This effect is seen in both the ECM and the Roofline model, and we believe it is linked to the undisclosed behavior of the L3 cache. The cache simulator apparently overestimates the number of L2 or L3 misses and predicts a higher main memory traffic volume. Unfortunately, AMD Zen does not have main memory traffic hardware performance counters, so we are unable to validate this assumption.

In light of the large main memory traffic contribution, we would suggest temporal blocking to bring the inverse throughput down to the  $T_{\text{RegL1}}$  level.  $T_{\text{L3Mem}}$  contains all memory accesses (i.e., transfers between main memory, L3 and L2).

---

```

double a[M][N][P], b[M][N][P];
double c0, c1, c2, c3, c4, c5, c6, c7, c8, c9;
double c10, c11, c12, c13, c14, c15, c16, c17, c18;

for(int k = 3; k < M-3; k++) {
    for(int j = 3; j < N-3; j++) {
        for(int i = 3; i < P-3; i++) {
            b[k][j][i] = c0 * a[k][j][i]
                + c1 * a[k][j][i-1] + c2 * a[k][j][i+1] + c3 * a[k-1][j][i]
                + c4 * a[k+1][j][i] + c5 * a[k][j-1][i] + c6 * a[k][j+1][i]
                + c7 * a[k][j][i-2] + c8 * a[k][j][i+2] + c9 * a[k-2][j][i]
                + c10 * a[k+2][j][i] + c11 * a[k][j-2][i] + c12 * a[k][j+2][i]
                + c13 * a[k][j][i-3] + c14 * a[k][j][i+3] + c15 * a[k-3][j][i]
                + c16 * a[k+3][j][i] + c17 * a[k][j-3][i] + c18 * a[k][j+3][i];
        }
    }
}

```

---

**Figure 8.** 3D stencil code with radius 3, constant heterogeneous coefficients, star structure and double datatype

#### 4.3. A Long-ranged Stencil on Skylake X (Intel Xeon Gold 6148)

The second example showcases a long-ranged heterogeneous star stencil on the Skylake X architecture (Intel Xeon Gold 6148), the stencil source code is shown in Fig. 8. It features more floating point operations compared to the previous kernel because of the heterogeneous coefficients classification, but also higher memory traffic due to the long stencil range (i.e., range = 3 or r3 classification). For brevity, only the stacked ECM prediction with layer conditions as cache predictor is shown in Fig. 7. All remaining information and graphs can be found on the INSPECT website [21].

Qualitatively, both—Roofline and ECM prediction—represent the measured performance behavior well. The Roofline model is a bit too optimistic, and the ECM model a bit too pessimistic. The reason for that is the new organization of the cache hierarchy in the Skylake microarchitecture, seen in Fig. 2b and discussed in Sec. 2.3. At the moment it is not possible to correctly model the data transfers between L2 and L3 cache, in combination with main memory. With the ECM model a worst case scenario is assumed, such that all data dropped or evicted from L2 is passed onto L3. Taking that into account and with better knowledge of the actual caching algorithms and heuristics, the ECM prediction would become faster and more closely match the measured data.

The layer condition analysis correlates very well with the measured data, and all relevant breaks (i.e., plateaus) can be seen. The slow performance in the beginning until  $120^3$  is again related to the high  $T_{\text{comp}}$  fraction and scalar loads of the remainder loop.

Data shown on the INSPECT website for full-socket thread scaling shows that the prediction fits perfectly to the measured data. Also cache blocking for the L2-3D layer condition works very well, due to the larger L2 cache in this architecture. In contrast to that, L3-3D cache blocking works very poorly, as is in accordance with Intel’s recommendations: “Using just the last level cache size per core may result in non-optimal use of available on-chip cache” [27] (p. 41).

Overall it can be said that except for the uncertainty with the L2-L3-MEM caching behavior, the applied Skylake machine model works well and gives accurate predictions.

Performance optimization potential is again predicted by the plateaus (for spatial blocking) and contributions (for temporal blocking). Spatial blocking to  $N_{\text{block}}^{1/3} < 300$  may increase per-

formance by up to 30%. Temporal blocking would only make sense, in comparison to spatial, if it is done in the L2 cache, stripping the two upper contributions off the non-overlapping ECM prediction and possibly hitting the instruction throughput bottleneck ( $T_{\text{comp}}$ ) at 40 cy/CL.

#### 4.4. Comparison of a Short-ranged Box Stencil on Broadwell and Skylake X

Finally, we present a comparison of Broadwell (Intel Xeon E5-2697v4) and Skylake X (Intel Xeon Gold 6148) with a short-ranged box stencil with heterogeneous constant coefficients, cf. Fig. 9. Compared to star stencils, box stencils need more loads and registers, which may have a large performance impact. In Figs. 10a and 10b, benchmark data and model predictions are shown. On the INSPECT website a complete list of graphs, data sets and modeling information may be viewed for Broadwell [19] and for Skylake X [20].

On Broadwell performance does not seem to be impacted by data traffic nor in-core execution ( $T_{\text{comp}}$ ). Reasons for the poor and almost constant performance across all grid sizes is a register dependency chain that is visible in the assembly code, to be seen on the INSPECT website under “Kernel Source Code” by clicking on the “Assembly Code” button, but undetected by IACA. This dependency chain slows down the execution so much, that all other effects are suppressed and the performance becomes independent of the grid size. Since the number of available registers has been doubled with the introduction of the Skylake X architecture, this disastrous effect is eliminated there. Instead, until  $300^3$  the in-core bottleneck  $T_{\text{comp}}$  dominates and limits the reciprocal throughput to  $\sim 60$  cy/CL. This prediction, originating from an IACA analysis, is obviously too pessimistic, since measurements show better performance compared to ECM and Roofline models. Looking into the IACA analysis, it only gives an explanation for 46 out of the 60 cycles and adds 14 cycles based on an unknown heuristic. Considering this, 46 cycles per cacheline would explain the measured performance much better and calls for a

---

```

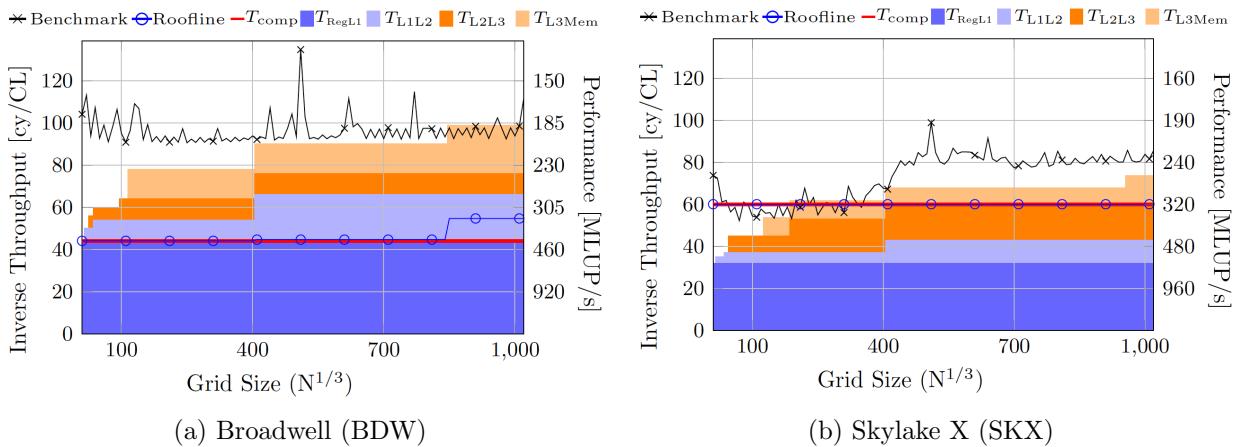
double a[M][N][P], b[M][N][P], w[27][M][N][P];

for(int k = 1; k < M - 1; ++k) {
    for(int j = 1; j < N - 1; ++j) {
        for(int i = 1; i < P - 1; ++i) {
            b[k][j][i] = w[0][k][j][i] * a[k][j][i]
                + w[1][k][j][i] * a[k-1][j-1][i-1] + w[2][k][j][i] * a[k][j-1][i-1]
                + w[3][k][j][i] * a[k+1][j-1][i-1] + w[4][k][j][i] * a[k-1][j][i-1]
                + w[5][k][j][i] * a[k][j][i-1] + w[6][k][j][i] * a[k+1][j][i-1]
                + w[7][k][j][i] * a[k-1][j+1][i-1] + w[8][k][j][i] * a[k][j+1][i-1]
                + w[9][k][j][i] * a[k+1][j+1][i-1] + w[10][k][j][i] * a[k-1][j-1][i]
                + w[11][k][j][i] * a[k][j-1][i] + w[12][k][j][i] * a[k+1][j-1][i]
                + w[13][k][j][i] * a[k-1][j][i] + w[14][k][j][i] * a[k+1][j][i]
                + w[15][k][j][i] * a[k-1][j+1][i] + w[16][k][j][i] * a[k][j+1][i]
                + w[17][k][j][i] * a[k+1][j+1][i] + w[18][k][j][i] * a[k-1][j-1][i+1]
                + w[19][k][j][i] * a[k][j-1][i+1] + w[20][k][j][i] * a[k+1][j-1][i+1]
                + w[21][k][j][i] * a[k-1][j][i+1] + w[22][k][j][i] * a[k][j][i+1]
                + w[23][k][j][i] * a[k+1][j][i+1] + w[24][k][j][i] * a[k-1][j+1][i+1]
                + w[25][k][j][i] * a[k][j+1][i+1] + w[26][k][j][i] * a[k+1][j+1][i+1];
        }
    }
}

```

---

**Figure 9.** 3D stencil code with radius 1, variable heterogeneous coefficients, box structure and double datatype



**Figure 10.** ECM and Roofline model predictions based on layer conditions, with benchmark results for 3D/1r/heterogenous/box/constant/double stencil on Broadwell and Skylake X architectures

better in-core model, as aimed for by the OSACA project [29]. Beyond  $N \approx 400^3$ , data transfers become more dominant and slow down the execution, as qualitatively predicted by the ECM model. Roofline sticks with the 60cy/CL, because no single memory level surpasses them. In light of the discrepancy between modeled and measured performance, the graph can not be used to guide performance optimization, but it sheds light on the IACA misprediction at hand.

For Skylake, simple spatial blocking with  $N_{\text{block}}^{1/3} < 400$  is advisable. Temporal blocking would not yield better results, because of the hard  $T_{\text{comp}}$  limit.

## 5. How to Make Use of INSPECT

When developing stencil-driven applications and especially when publishing performance results based on stencil codes, authors have to compare to a suitable, well-understood baseline. In order to make use of INSPECT in this context, the user must first classify their stencil according to our scheme and select a microarchitecture and CPU model from the INSPECT website that is similar or identical to their own. If that is not possible, INSPECT provides the toolchain and automation to compile a new baseline for future reference.

Depending on the programming language and software architecture, stencil patterns in applications may be hidden under several abstraction layers but come to light during detailed performance analysis. It is also the user's task to isolate the stencil code in order to be able to measure its performance. This may be done either "in situ" via suitable instrumentation or by writing a proxy application that only performs stencil updates. Once a stencil is classified and a comparison is established, optimization strategies may be guided by the INSPECT ECM model report: Spatial blocking should bring the performance of large data sets on the level of smaller data set sizes by better use of caches (moving to a plateau left in the plots), whereas temporal blocking strategies eliminate data transfers to lower memory hierarchy levels (peeling off layers in the stacked ECM plot contributions).

If the measured stencil performance in the application code does not coincide with the INSPECT data and model at least qualitatively, as seen in Sec. 4.4, the culprit is usually the compiler not generating efficient code, but other scenarios are possible: specific hardware features in the user’s benchmarking setup (e.g., different DIMM organization), unfavorable sys-

tem settings (e.g., small OS pages, uncontrolled clock speed, Cluster-on-Die settings, disabled prefetchers), simple benchmarking mistakes such as wrong or no thread affinity, etc. Whatever the reason, it will be worth investigating, which usually leads to better insight.

## 6. Related Work

Our work comprises three parts: stencil classification and generation with STEMPEL, benchmarking and modeling with Kerncraft, and presentation and publication of results on the INSPECT website.

Collecting and presenting benchmark results is a common approach for a variety of reasons. To name a few examples:

- TOP500 [34] ranks the performance of HPC systems world-wide based on the High Performance LINPACK [4] benchmark performance.
- HPCG benchmark [5] takes the same approach as the TOP500, with a different benchmark.
- SPEC [37] has a spectrum of benchmarks suites for different aspects and allows its members to publish the results on their website. Their suites come with real applications embedded as test cases. They produce detailed reports on the runtime environment, with the goal of comparing the performance of systems.
- STREAM benchmark [33] is the de facto standard for measuring main memory bandwidth. The website has results for machines in tabulated form.
- HPC Challenge Benchmark Suite [30] combines multiple benchmarks and allows users to publish results through HPCC's website.

All of these benchmark collections are focused on comparing machine performance by a set of predefined benchmarks, which is extremely valuable for purchasing decisions and as a reference for researchers and developers. In contrast, we try to explain the observed performance based on the characteristics of the stencil structure, which is usually defined by the underlying model and discretization. This makes it more informative and adaptable for a particular developer to compare and explain their own code's performance with similarly structured reference implementations provided by our framework.

The Ginkgo Performance Explorer [2] focuses on presenting performance data gathered by automatic benchmarks as part of a continuous integration (CI) pipeline. This project is generically applicable to other workflows, but lacks the focus on a specific field to allow the fine grained presentation of model predictions and measurements as is done by INSPECT, nor does it comprise any modeling component.

Methodologies for performance analysis most often fall into the category of performance models, such as the already mentioned ECM and Roofline models. Their application to specific stencils or stencil-based algorithms was at the focus of intense research [3, 7, 26, 31, 36]. Our concept goes beyond these approaches in that it enables easy reproduction of performance numbers and encourages discussion via an open workflow.

Datta et al. published a study of a stencil kernel on multiple architectures in 2009 [3]. It is based on the same modeling principles but does not provide a unified process and presentation reusable for other kernels.

Our approach also goes beyond known solutions to performance analysis and co-design in various application fields (see, e.g., [8, 35], and references therein), which usually concentrate on very specific stencil structures but lack our rigorous focus on reproducibility and modeling.

## Conclusion and Outlook

We have presented a comprehensive code generation, performance evaluation, modeling, and data presentation framework for stencil algorithms. It includes a classification scheme for stencil characteristics, a data collection methodology, automatic analytic performance modeling via advanced tools and a publicly available website that allows to browse results and models across a variety of processor architectures in a convenient way.

The presented baseline performance and model data provides valuable insight and points of comparison for developers who have to write performance critical stencil code. The automatic spatially optimized version is given as an optimization example. INSPECT already contains a large range of different stencil parameters and will be continuously extended to eventually comprise a full coverage of the parameter space. To this end, we plan on optimizing the tool chain to reduce the total runtime considerably. The choice of an analytic performance model over machine learning was deliberate as not only prediction but also insight into bottlenecks is desired.

Kerncraft’s support of non-Intel architectures is still rudimentary. Support for AMD’s latest x86 implementations is already available, and we have presented its preliminary use, while ARM will require more effort but is on our shortlist of upcoming features. These additions will be integrated into future updates of the INSPECT website.

An interesting spin-off of this work would be the integration of more web-enabled tools, such as the layer-condition analysis [11], into INSPECT to allow users to interactively analyze their own code. Compiler explorer [9] would be one potential tool to inspect compiler behavior for different architectures.

A generalization from stencils to dense linear algebra and streaming kernels is straightforward from Kerncraft’s perspective, but the classification scheme would have to be extended.

## Acknowledgements

The authors gratefully acknowledge funding by the German Federal Ministry for Education and Research (BMBF) via the SeASiTe, SKAMPY, and METACCA projects. We are thankful for support by the Bavarian Competence Network for Scientific High Performance Computing (KONWIHR).

*This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.*

## References

1. Abel, A., Reineke, J.: Uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 673–686. ASPLOS ’19, ACM, New York, NY, USA (2019), DOI: 10.1145/3297858.3304062
2. Anzt, H., Cojean, T., Flegar, G., Grütmacher, T., Nayak, P., Ribizel, T.: An Automated Performance Evaluation Framework for the GINKGO Software Ecosystem. In: 90th Annual

Meeting of the International Associate of Applied Mathematics and Mechanics, GAMM 2019, February 2019, Vienna, Austria. GAMM (2019)

3. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review* 51(1), 129–159 (2009), DOI: 10.1137/070693199
4. Dongarra, J.: The LINPACK Benchmark: An Explanation. In: *Proceedings of the 1st International Conference on Supercomputing*, Athens, Greece, June 812, 1987. pp. 456–474. Springer-Verlag, London, UK, UK (1988), <http://dl.acm.org/citation.cfm?id=647970.742568>
5. Dongarra, J., Heroux, M.A., Luszczek, P.: High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30(1), 3–10 (2015), DOI: 10.1177/1094342015593158
6. Fog, A.: Instruction tables (2018), [https://www.agner.org/optimize/instruction\\_tables.pdf](https://www.agner.org/optimize/instruction_tables.pdf)
7. Ghysels, P., Vanroose, W.: Modeling the Performance of Geometric Multigrid Stencils on Multicore Computer Architectures. *SIAM Journal on Scientific Computing* 37(2), C194–C216 (2015), DOI: 10.1137/130935781
8. Glinsky, B., Kulikov, I., Snytnikov, A., Romanenko, A., Chernykh, I., Vshivkov, V.: Co-design of parallel numerical methods for plasma physics and astrophysics. *Supercomputing Frontiers and Innovations* 1(3) (2015), DOI: 10.14529/jsfi140305
9. Godbolt, M.: Compiler Explorer (2019), <https://godbolt.org/>, accessed: 2019-09-06
10. Guerrera, D.: STEMPEL: Stencil TEMPlate Engineering Library (2019), <https://github.com/RRZE-HPC/stempel>, accessed: 2019-09-06
11. Hammer, J.: Layer Conditions: Interactive Web Calculator (2017), <https://rrze-hpc.github.io/layer-condition/>, accessed: 2019-09-06
12. Hammer, J.: pycachesim: Python Cache Hierarchy Simulator (2018), <https://github.com/RRZE-HPC/pycachesim>, accessed: 2019-09-06
13. Hammer, J.: OoO Instruction Benchmarking Framework on the Back of Dragons (poster). In: *The International Conference for High Performance Computing, Networking, Storage and Analysis*, SC18, November 2018, Dallas, Texas, USA (2019), [https://sc18.supercomputing.org/proceedings/src\\_poster/src\\_poster\\_pages/spost115.html](https://sc18.supercomputing.org/proceedings/src_poster/src_poster_pages/spost115.html)
14. Hammer, J., Eitzinger, J., Hager, G., Wellein, G.: Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. *Tools for High Performance Computing 2016*, Stuttgart, Germany, October 2016 pp. 1–22 (2017), DOI: 10.1007/978-3-319-56702-0\_1
15. Hirsch, I., Gideon S. (sic): Intel Architecture Code Analyzer (2017), <https://software.intel.com/en-us/articles/intel-architecture-code-analyzer>, accessed: 2019-09-06
16. Hofmann, J., Alappat, C.L., Hager, G., Fey, D., Wellein, G.: Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors, <https://arxiv.org/abs/1907.00048>, in Review/Pre-print.

17. Hofmann, J., Hager, G., Fey, D.: On the Accuracy and Usefulness of Analytic Energy Models for Contemporary Multicore Processors. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) Proceedings of the 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24–28, 2018. pp. 22–43. Springer International Publishing, Cham (2018), DOI: 10.1007/978-3-319-92040-5\_2
  18. Hofmann, J., Hager, G., Wellein, G., Fey, D.: An Analysis of Core- and Chip-Level Architectural Features in Four Generations of Intel Server Processors. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) Proceedings of the 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017. pp. 294–314. Springer International Publishing, Cham (2017), DOI: 10.1007/978-3-319-58667-0\_16
  19. Hornich, J., Hammer, J.: INSPECT: 3D Box Stencil on Broadwell, [https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/heterogeneous/box/constant/double/BroadwellEP\\_E5-2697\\_CoD/](https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/heterogeneous/box/constant/double/BroadwellEP_E5-2697_CoD/), accessed: 2019-09-06
  20. Hornich, J., Hammer, J.: INSPECT: 3D Box Stencil on Skylake X, [https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/heterogeneous/box/constant/double/SkylakeSP\\_Gold-6148\\_SNC/](https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/heterogeneous/box/constant/double/SkylakeSP_Gold-6148_SNC/), accessed: 2019-09-06
  21. Hornich, J., Hammer, J.: INSPECT: 3D Long Range Stencil on Skylake X, [https://rrze-hpc.github.io/INSPECT/stencils/3D/r3/heterogeneous/star/constant/double/SkylakeSP\\_Gold-6148\\_SNC/](https://rrze-hpc.github.io/INSPECT/stencils/3D/r3/heterogeneous/star/constant/double/SkylakeSP_Gold-6148_SNC/), accessed: 2019-09-06
  22. Hornich, J., Hammer, J.: INSPECT: 3D Short Range Stencil on AMD Zen, [https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/homogeneous/star/constant/double/Zen\\_EPYC-7451/](https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/homogeneous/star/constant/double/Zen_EPYC-7451/), accessed: 2019-09-06
  23. Hornich, J., Hammer, J.: INSPECT: 3D Short Range Stencil on Haswell, [https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/homogeneous/star/constant/double/HaswellEP\\_E5-2695v3\\_CoD/](https://rrze-hpc.github.io/INSPECT/stencils/3D/r1/homogeneous/star/constant/double/HaswellEP_E5-2695v3_CoD/), accessed: 2019-09-06
  24. Hornich, J., Hammer, J.: INSPECT: Intra Node Stencil Performance Evaluation Collection, <https://rrze-hpc.github.io/INSPECT>, accessed: 2019-09-06
  25. Hornich, J., Hammer, J.: INSPECT: Machine Models, <https://rrze-hpc.github.io/INSPECT/machinefiles>, accessed: 2019-09-06
  26. Hornich, J., Pflaum, C., Hager, G.: Efficient optical simulation of nano structures in thin-film solar cells. Computational Optics II, SPIE Optical Systems Design, Frankfurt, Germany, 14–17 May, 2018 (2018), DOI: 10.1117/12.2312545
  27. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual (2019), <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>, accessed: 2019-09-06
  28. Kulikov, I., Chernykh, I., Tutukov, A.: A new hydrodynamic code with explicit vectorization instructions optimizations that is dedicated to the numerical simulation of astrophysical gas flow. I. Numerical method, tests, and model problems. The Astrophysical Journal Supplement Series 243(1), 4 (2019), DOI: 10.3847/1538-4365/ab2237
-

29. Laukemann, J., Hammer, J., Hofmann, J., Hager, G., Wellein, G.: Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Dallas, Texas, USA, December 12, 2018. pp. 121–131 (2018), DOI: 10.1109/PMBS.2018.8641578
30. Luszczek, P., Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., Mccalpin, J., Bailey, D., Takahashi, D.: Introduction to the HPC Challenge Benchmark Suite (2005) (2005), <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.1817>, accessed: 2019-09-06
31. Malas, T., Hager, G., Ltaief, H., Stengel, H., Wellein, G., Keyes, D.: Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates. SIAM Journal on Scientific Computing 37(4), C439–C464 (2015), DOI: 10.1137/140991133
32. Malas, T.M., Hager, G., Ltaief, H., Keyes, D.E.: Multidimensional Intratile Parallelization for Memory-Starved Stencil Computations. ACM Trans. Parallel Comput. 4(3), 12:1–12:32 (2017), DOI: 10.1145/3155290
33. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)
34. Meuer, H.W., Strohmaier, E., Dongarra, J., Simon, H.D.: The TOP500: History, Trends, and Future Directions in High Performance Computing. Chapman & Hall/CRC, 1st edn. (2014)
35. Mitchell, N.L., Vorobyov, E.I., Hensler, G.: Collisionless stellar hydrodynamics as an efficient alternative to n-body methods. Monthly Notices of the Royal Astronomical Society 428(3), 2674–2687 (2012), DOI: 10.1093/mnras/sts228
36. Schäfer, A., Fey, D.: A Predictive Performance Model for Stencil Codes on Multicore CPUs. High Performance Computing for Computational Science – VECPAR 2012, Kobe, Japan, July 17–20, 2012 pp. 451–466 (2013), DOI: 10.1007/978-3-642-38718-0\_40
37. SPEC: Standard Performance Evaluation Corporation (2019), <https://www.spec.org>, accessed: 2019-09-06
38. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. Proceedings of the 29th ACM on International Conference on Supercomputing - ICS ’15, Newport Beach, California, USA, June 8-11, 2015 (2015), DOI: 10.1145/2751205.2751240
39. Treibig, J., Hager, G., Wellein, G.: LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In: 2010 39th International Conference on Parallel Processing Workshops, San Diego, California, USA, 13-16 September, 2010. pp. 207–216 (2010), DOI: 10.1109/ICPPW.2010.38
40. Williams, S., Waterman, A., Patterson, D.: Roofline. Communications of the ACM 52(4), 65 (2009), DOI: 10.1145/1498765.1498785

# Supercomputer Docking

**Alexey V. Sulimov<sup>1,2</sup>, Danil C. Kutow<sup>1,2</sup>, Vladimir B. Sulimov<sup>1,2</sup>**

© The Authors 2019. This paper is published with open access at SuperFri.org

This review is based on the peer-reviewed research literature including the author's own publications devoted to supercomputer docking. The general view on docking and its role at the initial stage of the rational drug design is presented. Molecules of medicine compounds selectively bind to the active site of a protein, which is responsible for the disease progression, and stop it. Docking programs perform positioning of molecules (ligands) in the active site of the protein and estimate the protein–ligand binding energy. The larger this energy is, the less concentration of the respective compound should be used to observe the desired effect. Several classical docking programs are described in short. Examples of the adaptation of existing docking programs to supercomputing and using them for virtual screening of millions of ligands are presented. Two novel generalized docking programs specially designed for multi-core docking of a single ligand on a supercomputer are described shortly. These programs find a sufficiently wide spectrum of low energy minima of a protein–ligand complex in the frame of a given force field. The quasi-docking procedure using the generalized docking program is described. Quasi-docking allows to perform docking with quantum-chemical semiempirical methods. Finally a summary is made based on the materials presented.

*Keywords:* docking, protein–ligand, global optimization, tensor train, force field, quantum-chemical method.

## Introduction

Docking programs are extremely demanded for the computer aided structural based drug design [78, 80]. The latter is used at the initial stage of the long (10–15 years) and expensive (> 1 billion USD) rational drug design pipeline. This initial stage is the cheapest of all the following stages: preclinical testing (animals), three phases of clinical stages (humans), approving and post approving stages, but the initial stage plays the key role in the whole pipeline success defining the diversity of the active compound–candidates, their selectivity and low toxicity. The main idea of the rational drug development is to find a compound, the molecules of which bind specifically to a definite region (active site) of the given bio-molecule, e.g. a protein, responsible for the disease progression, and stop or change the latter. The larger this binding energy is, the lesser concentration of the active compound should be used to achieve the desired effect. Docking programs carry out positioning of the molecules–candidates (ligands) in the definite site of the target protein and calculate the protein–ligand binding free energy [95] which is directly connected with the measured binding (dissociating) constant of the active compound. There are several dozen of docking programs and a dozen of Internet docking sites available either for free or on a commercial basis [14, 66]. Despite the obvious progress of the docking technique and plenty of success stories, there are still various challenges [9, 96, 107]. While in many cases positioning accuracy of docking is satisfactory, the accuracy of binding energy calculations is insufficient to perform the hit-to-lead optimization on the base of docking results. The increase of docking accuracy should result in an increase in the effectiveness of the whole process of new drug development [90].

The work of many docking programs relies on the docking paradigm which assumes that the ligand binds in the active site of the target protein at the global energy minimum of the

<sup>1</sup>Research Computer Center of Lomonosov Moscow State University, Moscow, Russian Federation

<sup>2</sup>Dimonta Ltd., Moscow, Russian Federation

protein–ligand system [60, 61]. Thus, the docking problem turns into the global optimization problem. The main challenge of docking is a high dimensionality and complicated relief of the energy surface where the global energy minimum should be found. For example, a therapeutic molecule often has 10 internal rotational degrees of freedom (torsions). Thus, the protein–ligand system energy surface has  $16 = 10 + 6$  dimensions where 6 dimensions correspond to translations and rotations of the ligand as a rigid body. This case corresponds to the rigid target protein. However, if protein flexibility is taken into account, the dimension of the protein–ligand energy surface increases in line with additional number of protein degrees of freedom, e.g. for every mobile protein atom additional 3 dimensions, atom Cartesian coordinates, should be added. On the other hand, such a docking program should be able to perform docking of many ligands for the reasonable period of time. This contradiction results in many different simplifications implemented in docking programs, especially at the dawn of the docking technique development 25 – 30 years ago when available computer resources were very limited. The “Faster, even faster” mantra long time was the main goal of docking algorithms and some of popular docking programs can dock one ligand at one processor for less than 1 minute even less, e.g. the ICM program can run docking of one ligand for 10 seconds [57]. However over time, more attention has been paid to the docking accuracy, and in some popular docking programs, for example in Glide, in addition of Standard Precision (SP) calculations the Extra Precision mode (XP) has been added [27], and the latter needs much more computing resources. If program docks one ligand at one computing core for the time of half an hour or longer, the use of this program for virtual screening of large databases containing many thousands of ligands inevitably needs supercomputer resources. On the other hand, either for the too complex form of the protein active site and respective complicated protein–ligand energy surface or for the too large number of ligand torsions it might be needed to increase considerably the thoroughness of the global energy minimum search at the expense of a large increase of the docking time of one ligand. This increase can be compensated by the multi-processor performance of the docking program at several dozen or hundred computing cores of a supercomputer. Additional processing of docking results can also demand supercomputer resources especially if the protein–ligand binding energy refinement needs an employment of molecular dynamics (MD) or quantum chemistry (QC) [51]. Actually MD methods have not been used for virtual screening of large ligand databases, but these methods of the protein–ligand binding free energy calculation are applied usually at the post-docking stage for a further refinement of the binding energy of best selected ligands and to optimize them. MD methods being applied to the binding free energy calculations have a lot of specific features and tricks [45], e.g. artificial lowering of energy barriers, and a detailed description of all of them needs a lot of space and it is out of the scope of this review. We only note here several points. First, MD methods and docking have some common features defining accuracy of the calculations and the most important one is the energy calculation method using either a force field or a quantum chemistry method. Second, calculations of the absolute protein–ligand binding free energy is a much more challenging problem than calculations of the relative binding energy and in most publications just the latter is calculated with MD [105]. Third, most of these MD methods sample the Boltzmann distribution of molecular configurations coupled to a heat bath at a given temperature, and same sampling can be done also with Monte Carlo simulations [16]. Fourth, there are several methods to perform the protein–ligand binding free energy calculations [46], and the most widely used ones are thermodynamic integration (TI) [28, 42, 53] and free energy perturbation (FEP) [19] methods, as well as the recently proposed enveloping

distribution sampling (EDS method [70, 75]; a recent survey of the state of art of the binding free energy calculation method is presented in [54] (see also a perpetual review of David L. Mobley, et al. at <https://github.com/MobleyLab/benchmarksets/blob/master/paper/benchmarkset.pdf>). Fifth, the protein–ligand binding affinity can be associated with the ratio of kinetically determined rates  $k_{off}$  and  $k_{on}$  of complex formation [44] and MD simulation of binding kinetics can become a useful tool to get insight of a mechanism of protein–ligand binding [67]. Sixth, MD penetrates into the docking technology in different ways from prior generation of an ensemble of protein conformations to take into account protein flexibility performing docking into discrete rigid pre-generated protein conformations (ensemble docking) to the so-called solvent mapping for the identification of binding sites and hot-spots on protein surfaces [18, 24, 30]. Some additional recent examples of ensemble docking can be found in [2, 3, 5, 22, 23, 82, 86, 106]. All these methods require large computational resources and supercomputing power should be demanded along this road.

Quantum chemistry (QC) plays an important role in modeling for drug design. However, for the energy calculation in docking proper QC methods have not been used until recently. The latter is connected with the inability of QC programs, based on *ab initio* including DFT methods, to perform calculations of molecular systems containing thousands of atoms. Even the model of a protein–ligand complex reduced to 5–6 hundred atoms containing only the ligand and the active site of the target cannot be used for docking with QC *ab initio* methods due to the time limitation and limited capacity of multi-processor performance. Nevertheless, some recently developed QC semiempirical methods together with the localized molecular orbitals method can be used for post-docking refinement of the energy of the different conformations of the protein–ligand system (see below the “Quasi-docking” section). The QM/MM approximation where the most important part of the molecular system is treated with QC methods (QM abbreviation) and the rest of the system is described at a level of classical molecular mechanics (MM abbreviation). For example, QM/MM was used in [4] for the post-docking optimization of the protein–ligand system using the old AM1 semiempirical method, and the best ligand poses were found at the preliminary step by docking with the AutoDock program. Despite the good results demonstrated in [4] eclecticism of this attempt are obvious because docking is performed in the frame of an empirical oversimplified AutoDock force field (a set of classical potentials describing inter- and intra-molecular interactions), MM part is calculated with another, Charmm, force field and for QM part the AM1 method badly describing hydrogen bonds and dispersion inter-molecular interactions [102] is used. Different approaches of QC used for the binding affinity refinement in post-docking processing can be found in [15, 41, 77]. So, QC-docking and QC molecular dynamics for the accurate protein–ligand binding free energy calculations are still waiting for their realization, and this will happen in near future due to supercomputer facilities. QC methods are also used either for creation of whole force fields or for improving existing force fields, by modifications of atomic charges on the base of QC calculations. The most widely recognized example of the latter is the GAFF force field [104] where atomic charges are calculated with QC, and Van der Waals parameters are taken from the empirical potentials. Some examples of this approach are also presented in [8, 103], but there are much more. However, we should note that atomic charges are the part of a whole set of force field parameters describing inter- and intra-molecular interactions, and these parameters must be obtained in a self-consistent manner.

The molecular models of the target protein and ligands are very important for the proper docking performance, as well as MD calculations. 3D structures of proteins and protein–ligand complexes stored in Protein Data Bank [6, 13] do not usually contain hydrogen atoms. Hydrogen atoms are added to the protein structures with the help of a particular programs. The resulted 3D structures obtained after such different programs performance are different from one another, and this influences strongly on docking [47] due to the difference in the spatial positions of the added hydrogen atoms and due to difference in the protonation states of some amino acids residues. The best way to define protonation state is based on QC calculations but usually this is done with standard rules for separated amino acids. Certainly, results of ligand docking also strongly depends on their protonation states, as well as on enantiomery, tautomerism, and in general QC methods (taking into account electron correlation at sufficiently high *ab initio* level) should be used for a proper preparation of the respective enantiomers and tautomers together with low energy stereoisomers of a given ligand. The latter is closely connected to the determination of the lowest energy conformation of the unbound ligand in water solution defining the ligand internal stress energy which is an important term of the binding free energy. Unfortunately, in many cases QC methods still are not used for these purposes. Some examples are presented in [10, 11, 40, 68].

Nowadays machine learning invades many areas of science, and docking is not an exception. For example, convolutional neural networks (CNNs), which are commonly used in the image recognition [50], are applied in [71] to the construction of the scoring function to discriminate between correct and incorrect ligand binding poses and known binders and nonbinders, and it is shown that the CNN scoring function outperforms the AutoDock Vina one. The more sophisticated use of deep learning is presented in [49] where RAVE method (the reweighted autoencoded variational Bayes for enhanced sampling) [74] is used for calculating absolute protein–ligand binding free energies. Other recent examples of machine learning application to docking and scoring are presented in [58, 59, 69, 99]. In connection with this dawn of new era of machine learning application we should make a remark that databases of experimentally measured protein–ligand binding affinities contain many hidden uncertainties and sometimes errors, and before their use for machine learning, these databases should be strongly and cleverly filtered.

Considering that reviews on classical (non–supercomputer) docking programs are published regularly [14, 66, 96, 107], we review here mainly works devoted either to development of supercomputer docking programs, or to the application of supercomputers to virtual screening of large databases of ligands and refinement of docking results. We restrict this review by only docking of small molecular weight ligands into proteins. The protein–protein docking is out of the scope of this work, but all problems discussed here especially protein flexibility and docking accuracy are key features of such docking programs. Two examples of a protein–protein docking program, supercomputer docking and webserver, are presented in [62, 110]. Firstly, we take a look at the docking problem and consider examples of classical docking programs and their peculiarities. Secondly, we present a review of publications on docking in conjunction with supercomputers. Finally, some concluding remarks are presented.

## 1. Docking: General View

For brevity we will call as inhibitors those molecules (ligands) that bind to a given target proteins. Inhibitors block functioning of the target proteins, but in general ligands upon binding can activate target proteins or change their work. Usually a low molecular weight ligand contains

several dozen atoms. Target proteins contain several thousand atoms, and 3D structures of many target proteins, Cartesian coordinates of all protein atoms, can be free downloaded from the Protein Data Bank (PDB) Web-site (<https://www.rcsb.org>) [6, 13]. This database contains now more than 150 thousand entries of biological macromolecular structures (protein, DNA, RNA) and their complexes with ligands.

Docking is used to discover non-covalent inhibitors of a given target protein. This means that a ligand and a protein are bound with one another by weak inter-molecular interactions, this binding is reversible and the protein-ligand binding free energy  $\Delta G_{bind}$  is directly related to the binding constant  $k_b$  associated with protein  $[P]$ , ligand  $[L]$  and their complex  $[PL]$  concentrations in water solvent:

$$k_b = \frac{[PL]}{[P] \times [L]} = e^{\frac{-\Delta G_{bind}}{RT}}, \quad (1)$$

where the dimensionality of the binding constant is  $mol/L$ ,  $R$  is the universal gas constant, and  $T$  is absolute temperature in  $K$ . The binding free energy is defined as  $\Delta G_{bind} = G_{PL} - G_P - G_L$ , where  $G_{PL}$ ,  $G_P$  and  $G_L$  are Gibbs free energies of the protein-ligand complex, the protein and the ligand, respectively. In its turn the Gibbs free energy of a molecular system  $X$  can be calculated through the respective configuration integral  $Z_X = -RT \ln(G_X)$  which is calculated over the phase space of the molecular system  $X$ . If the low energy relief of the molecular system  $X$  is known, the configuration integral can be calculated. Moreover, if the global energy minimum is notably (comparing with  $kT$ ) stands out by its energy in respect with energies of other low energy minima, the molecular system spends most time in the global minimum, and the configuration integral and corresponding Gibbs free energy of the system is defined by only the global energy minimum value and by the form of the respective energy well.

In many cases target proteins can be considered as rigid systems and the global energy minima search should be performed for only the protein-ligand complex and for the unbound ligand while the energy of the unbound protein is calculated for only one its configuration defined by atom coordinates taken for PDB. PDB-files usually contain only coordinates of heavy (non-hydrogen) atoms. Several thousand hydrogen atoms should be added to the protein structure using the appropriate program. Note that different programs add hydrogen atoms by different ways which result, first, in different protonation (charge) states of some molecular groups, the so-called titrated groups, and, second, in various hydrogen atoms positions, i.e. the covalent bonds of these hydrogen atoms can be differently oriented in space. These differences in spatial positions of added hydrogen atoms lead to different docking results and to different results of binding energy calculations [47].

One of the most important challenge of docking is the choice of the method of the molecular energy calculation which should include both inter- and intra-molecular interactions, as well as take into account the interaction of molecules with the water solvent. Quantum mechanics ( $QM$ ) or, what is the same, quantum chemistry should be used to describe molecular interactions properly. However, these methods could not be used until recently for the molecular systems containing several thousand atoms. Instead  $QM$  methods classical potentials, force fields, describing interatomic interactions are used. There are plenty of force fields created for the classical molecular dynamics simulations and docking: AMBER [104], CHARMM [7], MMFF94 [32], OPLS-AA [39], etc., which describe molecular interactions sufficiently satisfactorily but not without some faults in different particular cases.

However, even such a dramatic simplification as a changeover from *QM* methods to force fields is insufficient for the realization of the docking procedure as a global energy minimum search. The calculation of all pairwise interactions between all ligand atoms and all protein atoms for every protein–ligand complex configuration in the frame of any force field mentioned above is computationally too costly. Apart from strong simplification of force fields a model of preliminary calculated grid of potentials is implemented in many docking programs: potentials of ligand probe atoms interactions with the all protein atoms are calculated and stored at nodes of the grid covering the active site of the target protein. Then, in the course of the global energy minimum search the energy of the ligand in any position in the active site of the protein is calculated as the sum of the grid potentials over all ligand atoms. This grid approximation gives a large acceleration because all resource-intensive operations are performed at the grid calculation stage before the global optimization. However, the grid approximation restricts docking and its accuracy in several aspects. Firstly, it is impossible to make sufficiently accurate local optimization of the protein–ligand energy with variation of ligand atoms positions. Secondly, it is impossible to take into account mobility of protein atoms in the docking procedure. Thirdly, water solvent plays an important role in molecular energy calculations due to the high dielectric constant of water (78.4 at the room temperature) and respective strong screening of Coulomb interactions; in order to avoid to take into account thousands of explicit water molecules, implicit solvent models are used instead. In the frame of these models water solvent is described by a homogeneous dielectric continuum surrounding a molecule of solute and the nonlocal interactions of atom charges of the solute molecule with polarization charges on the solvent excluded surface (SES) cannot be sufficiently accurately reduced to predetermined local potentials at grid nodes. Fourthly, the fast increase of computing power will soon lead to the opportunity to utilize *QM* methods in docking, but such self-consistent methods cannot be used in the frame of the grid approximation.

The will to avoid the grid approximation, to use implicit solvent models, to take into account flexibility of ligand and protein simultaneously and by this to increase the docking accuracy leads to necessity to draw larger computer power for docking of one ligand and to the transition from notebook docking to supercomputer docking. Before the discussion of supercomputer docking in the next section we describe shortly some examples of popular classical docking programs to outline their limitations and to contrast them with supercomputer docking programs. The more or less exhaustive reviews of such docking programs are presented in [14, 66, 96, 107].

### **1.1. AutoDock**

This is the most popular docking package [14]. It is a free opensource code actually including two main docking engines employing a pre-calculated grid of potentials of probe ligand atoms interactions with the target protein: AutoDock Vina [98] and AutoDock [26, 56], and also auxiliary programs Raccoon2 (a graphical tool), AutoDockTools for the preparation of molecules and for the analysis of docking results and AutoLigand for predicting optimal sites of ligand binding. AutoDock Vina and AutoDock use stochastic methods for the global energy minimum search and the docking algorithms work mainly for the rigid protein model, although a possibility of taking into account protein flexibility is also implemented in AutoDock [55]. However the latter can be used only for very limited number of degrees of freedom: ligands should have less than 10 torsions and additional six degrees of freedom of two protein side chains. Actually ligand flexibility is described by only ligand torsions while bond lengths and valence angles are kept

fixed. AutoDock Vina uses an oversimplified empirical energy function and the conformational search is based on a gradient local optimization method. The energy function of AutoDock Vina does not contain the Coulomb term and consequently it considers electrostatic interactions only implicitly through the hydrophobic and the hydrogen bonding terms, it uses spherical potentials for hydrogen bonds and treats hydrogen atoms only implicitly. AutoDock uses more sophisticated but also a simplified empirical force field including directed hydrogen bond terms, explicit polar hydrogen atoms and Coulomb and van der Waals interactions and desolvation effects are treated in a very simplified local potential form. AutoDock uses the Lamarckian genetic algorithm for the global energy optimization [55]: a population of trial individuals, ligand conformations, is created, and then in successive generations these individuals mutate, exchange conformational parameters through the crossover procedure, and compete on their energy selecting individuals with lowest energy, and the “Lamarckia” feature allows individual to search their local conformational space, finding local minima, and then pass this information to later generations.

## 1.2. DOCK

This is the oldest docking program which was created more than 35 years ago. The initial version of this program perform rigid docking by matching ligand and receptor shapes on a steric overlap. Subsequently a force field, the energy minimization, solvent, ligand and protein flexibility, multiprocessor performance were implemented in it [1, 12]. The orientation search of ligand poses is performed by matching ligand heavy atoms to matching points inside the docking domain. The matching points are generated by filling the docking domain with spheres of varying radii ( $1.4 - 2.5 \text{ \AA}$ ); centers of these spheres become matching points. The preliminary calculated grid of potentials of probe ligand atom interactions with all protein atoms is used. There are different energy (scoring) functions: from simplest bump filtering that counts the number of van der Waals clashes between ligand and protein atoms and discard ligand poses that exceed a predetermined threshold to the energy score defined by Lennard–Jones and Coulomb potentials of the AMBER force field together with desolvation scoring based on the Generalized Born or the Poisson–Boltzmann implicit solvent models for the polar solute–solvent interaction plus the solvent accessible surface area (SASA) approximation for the non–polar term of the desolvation energy. The latter is a linear form of SASA. In the process of finding low–energy minima DOCK6 uses a combination of global sampling and a local energy minimization performed by the simplex method: rigid docking, including docking of different rigid ligand conformers, plays a central role in the performance of DOCK6, and the found ligand poses can be used for the further local energy minimization and *MD* simulation. The Anchor–and–Grow algorithm for flexible ligand docking is a distinguishable core feature of the DOCK6 program. A ligand is disassembled into non–flexible fragments connected by rotatable bonds. The largest fragments (anchors) are used to generate a set of possible anchor poses within the binding site. From these poses of each anchor a search for possible conformations of the ligand flexible parts is performed, and rigid fragments are flexibly attached to the anchor in layers until the whole ligand is reconstructed. When each ligand fragment is grown, the conformer with minimal energy is selected before proceeding to the next growing step. A key advantage of this algorithm is that the ligand conformation is left within the confinements of the binding site, radically reducing the conformational space of the search.

### 1.3. SOL

The main features of this program are [91, 95]: (i) rigid protein model; (ii) protein active site is represented by a grid of potentials describing interactions (electrostatic, van der Waals interactions) of all protein atoms with all possible types of ligand probe atoms in the frame of the MMFF94 force field practically without serious simplifications; (iii) inclusion of desolvation effects on the base of a simplified form of Generalized Born approximation [76, 85] allowing to store the corresponding desolvation potentials at nodes of the preliminary calculated grid; (iv) genetic algorithm is used for the global energy optimization; (v) ligand stress energy is taken into account in the frame of MMFF94 during the global optimization, i.e. the objective energy function of the global optimization consists of two terms: the ligand energy in the field of all protein atoms and the ligand stress energy. The latter is the energy of the ligand deformation. The desolvation energy is defined by the difference of the solvation energy of the bound protein–ligand complex and solvation energies of the unbound protein and ligand. The effect of desolvation is due to the fact that when a ligand binds to a protein, some of the surface protein and ligand atoms stop interacting with the solvent, water.

The docking region is defined by the position of the grid of preliminary calculated potentials. The grid is created in the cube (the docking cube) covering the whole protein active site. By default the cube edge is equal to  $22\text{ \AA}$ , and the cube contains  $101 \times 101 \times 101$  uniformly distributed grid nodes. The size and the position of the cube center are user defined. During docking performed by the global optimization a ligand can be randomly positioned at any pose inside the docking cube and it can have any conformation defined by random variations of torsions keeping fixed ligand bond lengths and valence angles. The grid is calculated by the SOLGRID module before the global optimization process. Opposite to many popular empirical force fields such as AMBER, CHARMM, OPLS–AA, the MMFF94 force field is considered as a physical *ab initio* force field because all its parameters are fitted to the results of *ab initio* quantum–chemical calculations for a broad set of organic molecules. MMFF94 has a well-defined procedure of the atom typification applicable to an arbitrary organic compound. This typification defines the force field parameters for any target protein and for almost any ligand. The atom typification, as well as the hydrogen atoms addition are made by the APLITE program. For SOLGRID calculations the MMFF94 atom typification is somewhat simplified up to 27 atom types (instead of 99 types) to keep the potentials in the computer RAM memory. So, the grid contains in its nodes 27 van der Waals and 27 Coulomb potentials of interactions of a respective ligand probe atom with all protein atoms and respective desolvation potentials. Note, that van der Waals potentials defined by equations of the MMFF94 force field undergo some broadening ( $\approx 0.3 - 0.5\text{ \AA}$ ) which can be defined in the set of SOLGRID input parameters. Broadening imitates restricted mobility of protein atoms. SOLGRID consumes from 1 to several hours at one computing core depending on the target protein size, and a binary file of about  $200\text{ Mb}$  is created. When the docking procedure is executed by the SOL module the ligand energy in the field of the protein is calculated as a sum of grid potentials of all ligand atoms corresponding to each ligand pose in the docking cube. A potential in the position of a given ligand atom is obtained by the interpolation of potentials in eight neighbouring grid nodes.

The SOL docking program works in accordance with the genetic algorithm as follows. First, the initial population of individuals (ligand poses in the docking cube) is created by random translations and rotations of a ligand as a rigid body and by a random generation of ligand torsions in their domains. The population size is the main input parameter of SOL, and it is

equal to 30000 by default. Then the total energy, the sum of the ligand grid and stress energies, is calculated for each of individuals and the latter are ranked in respect with their total energies. The individuals (ligand poses) with lowest values of the total energy are selected into the so-called mating pool where they take part in the creation of the population of the next generation through the crossover, mutations and direct translation keeping the population size fixed. Several individuals with the lowest energies are translated to the next generation without any change to be sure that the best ligand poses will not be lost in the next generation. Niching is used to choose diverse individuals in the mating pool. The niching procedure is implemented as a positive energy penalty given to a ligand pose which is close to the ligand pose that has been already chosen into the mating pool. Niching helps to avoid the degeneration of the population when all individuals are collected in one local energy minimum. After a given number of generations (1000 by default) the genetic algorithm stops and the individual (the ligand pose) with the lowest energy is taken from the last generation. This is the solution of the global optimizations problem: finding the ligand pose with the lowest total energy. Several (50 by default) independent runs of the genetic algorithm are performed to get some confidence in the reliability of the global optimization. All 50 ligand poses are clustered in respect with their positions: two ligand poses belong to one cluster if RMSD between their corresponding atoms is less than 1 Å. If population of the first cluster containing the ligand poses with lowest energies is sizeable and the number of clusters is small, the ligand pose with the lowest energy from the first cluster is considered a reliable solution of the global optimization problem. If there are many clusters and there is only one ligand pose in the first cluster, the solution of the global optimization problem is considered as unreliable and optimization procedure should be repeated with higher parameters of the genetic algorithm, mainly with larger population size and with more generations. SOL executes docking of one ligand with default genetic algorithm parameters for several hours at one computing core depending on the number of ligand atoms. First application of SOL in drug design has been made during discovery of new thrombin inhibitors [79]: more than 6000 ligands have been docked by SOL using X–Com grid technology, developed at the Research Computer Center of Moscow State University [25, 81]. Later the MPI multi–processor implementation of SOLGRID and SOL programs [91] make it possible to generate the grid of potentials, as well as to dock one ligand for less than 1 minute using several dozen computing cores. On the other hand, for the virtual screening of large ligand databases (dozens and hundreds of thousands of molecules) SOL can run on the Lomonosov supercomputer [100] distributing ligands over hundreds and thousands computing cores and docking one ligand per one core. Certainly some auxiliary scripts and programs are created to queue up respective jobs and to analyze the docking results. There is a number of success stories of the application of these three docking programs and other ones in computer aided structural based drug design and some of them are presented in [96].

## 2. Supercomputer Docking Programs

In spite of existence of many docking programs, only a relatively small subset of molecular docking codes have been ported to use many–core high performance architectures, such as GPUs or Intel’s Xeon Phi. The employment of supercomputer for docking pursues two main goals: to screen as many ligands as possible during a short period of time, and to dock one ligand as accurately as possible. The increase effectiveness of these multi–processor calculations is also an important technical problem.

## 2.1. Faster and Larger

Modern chemistry opens its almost unlimited space of small molecular weight ligands for drug design creativity. Millions of on-shelf compounds are now available for different libraries and databases. Much more virtual molecules can be generated, but the possibility of their synthesis is questionable in many cases. All these ligands can be used as a starting pool of candidates for a given, especially new, target protein at the initial stage of the drug development giving a diversity of hit compounds and their optimization results in a diversity of lead compounds. In this section we present several examples of adaptation of existing docking programs to supercomputers for the virtual docking screening of thousands and millions of ligands.

## 2.2. HSP–DOCK

One of the challenges of supercomputer docking is to create the infrastructure that would allow anyone the ability to dock as many ligands as possible and as quickly as possible. One solution of this problem is presented in [97] where using profiling optimization with the help of a specially designed software “wrapper” called HSP–DOCK authors are able to dock large numbers of ligands efficiently with the widely used DOCK6 program on a large, non-shared memory NICS Kraken supercomputer system: a cluster computer made up of SMP (symmetric multi-processor) nodes [73]. When compared with the standard distribution of MPI multi-threading with DOCK6, HSP–DOCK showed near-linear scaling to the number of cores available to it, without necessitating the modification of the DOCK algorithm code itself. Here are some details of this docking experiment. Four different target proteins were chosen. A library of 1.4 million lead-like molecules from ZINC database was docked against each target. In this case lead-like compounds have the number of torsions  $\leq 7$ , and molecular weight of less than 350 Da. The targeted site of each protein was based on known biological interactions and refined further using MOE’s site finder utility providing precise docking decoys along with propensity for ligand-binding scores. The resulting dataset was analyzed for electrostatics (*ES*), van der Waals (*VDW*) and GRID (the summation of *ES* + *VDW*) scores. Docking is benchmarked on two different computer setups including a small academic cluster and a massively parallel supercomputer: the Medical University of South Carolina Computational Biology Resource Center (MUSC CBRC using MPI–DOCK) cluster of 46 nodes and the National Institute of Computer Science (NICS) Kraken Cray XT5 with 112896 cores (using HSP–DOCK). It is found that the DOCK6 standard MPI version does not scale well on tens of thousands of cores, or when the library of ligands grows into the millions. MPI scaling becomes inefficient because I/O contention and saturation of the master process causes the cluster to spend more time waiting for I/O tasks than carrying out computations. HSP–DOCK dynamically schedules workload distribution to individual nodes to optimize output routines, reducing I/O thrashing and increasing time spent in computation. The MUSC CBRC mean time to dock one ligand per one core was 88 s to be compared with 20 s on Kraken. It takes  $\sim 17$  h using 32000 cores of Kraken to dock 50 million flexible ligands into a single target protein. However such large scale docking leads to the need of the posterior analysis of the docking results. Examples presented in this study show that after docking of 1213412 compounds into one target protein there are 57017 hits, i.e. ligands with a score better than a specified threshold. For other investigated targets the number of hits is of the same order: 1 – 5 % of all successfully docked compounds. The necessity of further elaborations of several dozen thousand hits either with more accurate docking program or with bioinformatics

becomes obvious. One may wonder to perform more accurate docking from the beginning, but it needs much more computing resources and another organization of such massive docking. So, the example presented in [97] demonstrates the possibility of fast and effective virtual screening of very large databases of compounds using the DOCK6 docking program without any changes of the code with the help of a software “wrapper” HSP–DOCK on the supercomputer Kraken Cray XT5 of the National Institute of Computer Science.

### 2.3. BUDE

Another example of a supercomputer multi–core docking program is the Bristol University Docking Engine (BUDE) [52]. The special thrill of BUDE is a high effective realization for a supercomputer performance and compatibility with different types of hardware. It is adapted for modern day accelerators and it has highly optimized computational kernels for the docking and scoring functions using OpenCL, which are capable of sustaining a significant fraction of the hardwares peak performance. Exploiting OpenCL enables performance portability across a wide range of different computer architectures, including CPUs, GPUs and accelerators. BUDE has undergone extensive optimization, and can sustain over 40 % of floating point peak performance on a wide range of different architectures. The main features of BUDE is the energy minimization algorithm which is based on the Evolutionary Monte Carlo (EMC) techniques described in [29]. This genetic algorithm based optimization is performed in the space of the six degrees of freedom – the rigid protein and ligand approximation and a tuned empirical free–energy force field is employed for predicting the binding pose and for estimating the protein–ligand binding energy. In the course of a single ligand conformation docking BUDE evaluates the binding energies for hundreds of thousands of ligand poses, and each pose is evaluated independently. The optimized version of BUDE is several times faster than the previous non–optimized version of BUDE on eight different hardware with either GPU or CPU: NVIDIA *GTX* 680, NVIDIA *GTX* 780 *Ti*, NVIDIA *Tesla* K20c, AMD *Radeon* HD7970, AMD *Radeon* R9290X, AMD *FirePro* S10000, Intel *Xeon Phi* SE10P and Intel *E5* – 2687W (*x2*). For example the optimizations developed for the *GTX* 780 *Ti* have demonstrated the biggest improvement of the Nvidia GPUs with a 4.5× increase over the baseline version of BUDE. The optimized BUDE code achieves 44 billion atom–atom *interactions/s* on the *GTX* 680, taking 37 s to dock the 128 conformations (~ 0.3 s per conformation). It achieved a sustained performance of 1.43 *TFLOPS* when measured across the entire BUDE run, representing 46 % of the peak single precision performance of the device. The obvious drawback of this work is a too simplified docking model but it shows a direction of possible improvements of performance of any docking program.

### 2.4. VinaLC

This program is specially designed for the multi–core supercomputer performance with the goal to carry out virtual screening of very large ligand databases by docking [108]. VinaLC is the realization of the AutoDock Vina docking program [98] that is enhanced by a mixed parallel scheme, where within each node multithreading is used, and across different nodes an MPI parallel scheme is applied. VinaLC was developed and tested on petascale supercomputers at Lawrence Livermore National Laboratory. It scales up to 15408 CPU cores demonstrating overheads of less than 4 % and it is able to dock 17 million flexible ligands on 15408 cores during 24 hours. The docking accuracy of VinaLC defined by a model implemented in AutoDock Vina

was investigated in [108] using the Directory of Useful Decoys (DUD) dataset [35] and the reassuring results were obtained.

AutoDock Vina was implemented and tested later [38] on four different HPC infrastructures available to academic researchers in the Netherlands: an 8 – *core* virtual machine on the Dutch Academic HPC cloud, a local cluster at the Academic Medical Center of the University of Amsterdam with 128 cores, the Hadoop cluster for scientific research consisting of 90 *data/compute* nodes (adding up to more than 1400 cores) and has a distributed file system with a capacity of 630 *TB*, and the Dutch eScience grid which includes a dozen PBS (portable batch system) clusters all around the Netherlands (>10000 cores). These HPCs were employed to perform virtual screening of about 100000 compounds from ZINC [36] and other libraries investigating parallelism, docking time, etc.

Some details of a more recent use of AutoDock Vina for virtual screening are presented in [20] (docking on the Salomon (Czech Republic) supercomputer) and in [33] (docking on the peta-flops-scale Anselm supercomputer <https://www.it4i.cz>). Another example of the adaptation of the docking program from AutoDock family was presented in [21], where AutoDock4.lga.MPI was used to dock 1 million compounds. Details of MPI-realization of the AutoDock4.lga.MPI program were presented in [17], but in [21] particular attention was paid to the optimal organization of docking of millions of ligands on thousands of processors when file preparation and result analysis define the effectiveness of the whole virtual screening job. Among others the clusterization of ligand poses found in independent runs plays an important role in the revelation of best pose and finding best binders among millions of candidate molecules. As a result 1 million ligands having from 0 to 32 torsions (but only 5 % of the database are ligands with at least 10 torsions) have been docked on the Jaguar *Cray XT5* Supercomputer at Oak Ridge National Laboratory using 65 thousand processors. This job took almost 24 hours, but nearly the whole database was screened in 10 hours and only docking of extremely flexible ligands increased the total time.

As predecessor of supercomputer docking, the grid technology should be mentioned. One example of such distributed docking calculations on large scale grids is presented in [37] where two docking programs, AutoDock and FlexX [72] were used for virtual screening millions of ligands against two targets to struggle with malaria and avian influenza.

## **2.5. More Accurate**

Some efforts have been undertaken in attempts to understand main reasons of unsatisfactory reliability of most of docking programs and to improve the docking accuracy. The quick increase of available supercomputer resources gives a fortunate opportunity for such investigations. We present here main features of two docking programs of a new generation and then their use as a tool for the employment of quantum-chemical methods for docking.

## **2.6. Generalized Docking Programs**

### **2.6.1. FLM**

FLM – massive parallel low energy minima search [48, 60, 92]. The MPI-based FLM docking program performs multi-processors local optimizations of the energy of the protein-ligand complex from random initial ligand positions and conformations. As a result the spectrum of all or almost all low energy minima is found. Among these minima the global energy minimum

should present and in respect with the docking paradigm the best ligand pose should correspond to this global minimum. The term “generalized” is used for this program because the result of its performance is not only the global energy minimum but the whole spectrum of low energy minima. Obviously, the good accuracy of ligand positioning is an indispensable prerequisite of high accuracy of the protein–ligand binding energy calculation. FLM works in the frame of the *ab initio* MMFF94 force field [32] without simplifications and a preliminary calculated grid of protein–ligand interaction potentials and without the use of fitting parameters. There are two versions of this docking program: faster *FLM*-0.05 where energy is calculated without taking into account the interaction of protein–ligand complex with water solvent, and slower *FLM*-0.10 where the PCM implicit solvent model is implemented. The initial ligand positions and conformations in the docking space are generated by random translations and rotations of the ligand as a rigid body and by internal rotations (torsions) of molecular groups around ordinary single bonds which are not included in the cyclic structures, bond lengths and values of valence angles are kept fixed. The local energy optimization is performed by the well-known gradient based L-BFGS method. The optimization is made with variation of Cartesian coordinates of all ligand atoms keeping all protein atoms fixed. The docking space is defined by a sphere of a given radius and the location restricting the position of the geometrical center of the ligand. The number of low energy minima kept in the docking procedure is defined by the input parameter which can be any integer number. The scalability of the FLM performance with the growth of the number of CPU cores is very good up to several thousand cores because each local optimization is performed on a single core. One of peculiarities of the FLM program is the time unlimited performance when the job time and the number of cores are restricted only by a supercomputer work organization. The goal of the FLM performance is not only to find the global energy minimum with high reliability but also to find a given number of low energy minima without skipping any minimum. Exercises with a test set of protein–ligand complexes where FLM was used for docking native ligands into the proteins with which these ligands were crystallized showed that as a rule the global energy minimum was found already after several dozen of thousands of local optimizations, but for the detection of several thousand low energy minima it was needed to perform several hundred thousand, sometimes (for ligands containing about 15 – 20 torsions) more than a million local energy optimizations and in total  $\sim 10^9$  energy calculations of the protein–ligand complex were needed for generalized docking of one ligand. It was demonstrated that *FLM*-0.05 could find the global energy minimum near the crystallized native ligand position only for a small portion of test complexes. However *FLM*-0.10, where the implicit solvent model was implemented, could find the global energy minimum near the crystallized native ligand pose for more protein–ligand complexes than *FLM*-0.05 could do. So, the use of implicit solvent models together with force fields is obviously better for good docking positioning. FLM can be exploited also for finding the low energy minima spectrum of the unbound ligand. This operation is needed for the determination of the global energy minimum of the unbound ligand defining the ligand stress energy which is an important term in the expression for the protein–ligand binding energy. It turned out that FLM could be used for the comparison of different force fields in the docking procedure and for the investigations of effectiveness of different docking algorithms. Special energy minima indexes have been introduced which are convenient for the description of the low energy minima spectrum of protein–ligand complexes [48, 60, 61, 92–94]. In particular, the index of the minimum Near Native (INN) is useful for the indication of the docking accuracy. Its meaning is as follows. All low energy minima

found by a generalized docking program can be ranked by their energy in the ascending order. After this ranking every minimum gets its own integer index which is equal to its number in this ranked list of minima. The lowest energy minimum obtains the index equal to 1. INN indicates the index of the minimum corresponding to the ligand pose differing from the crystallized native ligand position with RMSD (the root mean-square deviation over all ligand atoms) less than  $2\text{\AA}$ . If the docking program finds several minima close to the crystallized native ligand position, INN will be equal to the index of the minimum with the lowest energy (the lowest index from all possible ones). When INN equals to 1, the docking paradigm is satisfied: the found global minimum of the energy of the protein–ligand complex is near the crystallized native ligand pose. FLM is a very resource-intensive docking program and this is its main drawback: it needs about  $10000 - 20000 \text{ CPU * hours}$  for generalized docking of one ligand depending on its size and flexibility.

### 2.6.2. SOL-P

SOL-P uses the tensor train global optimization algorithm [61, 93, 94]. SOL-P is also the generalized docking program. It performs the search for the global energy minimum, as well as for other low energy minima of the protein–ligand complex in the frame of the MMFF94 force field [32] without any simplifications and without using fitting parameters; it does not use neither the preliminary calculated grid of potentials of ligand probe atom interactions with the target protein, nor premeditated points of ligand positioning in the active site of the protein. The main peculiarity of this program is the novel tensor train (TT) docking algorithm. This algorithm is more effective than the widely used popular genetic algorithm [109]. SOL-P docks flexible ligands into the rigid protein (the early versions of SOL-P have been designated as SOL-T [61]), as well as into the protein with moveable atoms. In the latter case the space for the global energy minimum search is described by the translations and rotations of the ligand as a rigid body, internal torsions of the ligand and by Cartesian coordinates of selected protein atoms. In the case of docking with moveable protein atoms the INON index (Index Near Optimized Native) has been introduced instead of INN [93, 94]. The definition of INON is almost the same as one of INN but the ligand pose corresponding to an energy minimum is compared by RMSD with the optimized native ligand pose obtained by the local energy optimization from the initial crystallized native ligand position. SOL-P for a rigid protein is much faster than FLM: SOL-P needs only about  $5 - 100 \text{ CPU * hours}$  for docking a flexible ligand depending on its size and the number of torsions, but FLM carries out a more thorough search of low energy minima. The effectiveness of finding of the global energy minimum by FLM and SOL-P is almost the same. TT docking algorithm is based on the recently developed methods of the tensor analysis and in particular on the TT global optimization algorithm. Tensors, multi-dimensional arrays, appear in the docking problem as follows. The energy of the protein–ligand complex in the frame of the MMFF94 force field is a continuous function depending on the variables describing all degrees of freedom ( $d$ ) of the molecular system. If the discretization of each of these variables is carried out by a set of discretization points ( $n$ ), e.g. equidistant points, the continuous energy function is transformed into  $d$ -dimensional tensor with  $n^d$  elements. If the discretization is sufficiently fine, the solution of the continuous global optimization problem coincides with the solution of the discrete global optimization problem. The TT global optimization algorithm determines the largest in magnitude element of a tensor. Docking is the global minimization problem but it can be easily converted into the global maximization problem applied for example to the functional:

$$f(x, E_*) = \exp^{100 \operatorname{arccot}(E(x) - E_*)} \quad (2)$$

where  $E(x)$  is the dimensionless MMFF94 energy for the given conformation  $x$  of the protein–ligand complex,  $E_*$  is the global minimum found on the previous iteration. The global maximization problem is solved on the base of TT–Cross method [65] which is used to obtain approximation of the tensor  $A(i_1, \dots, i_d)$  in the tensor train (TT) representation:

$$A(i_1, \dots, i_d) \approx \sum_{\alpha_1=1, \dots, \alpha_{d-1}=1}^{r_1, \dots, r_d} G_1(i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \dots G_{d-1}(\alpha_{d-2}, i_{d-1}, \alpha_{d-1}) G_d(\alpha_{d-1}, i_d). \quad (3)$$

The numbers  $r_1, \dots, r_{d-1}$  are called TT–ranks of the tensor. The 3–dimensional tensors  $G_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$  are called cores or carriages of the tensor train. Operations on tensors in the TT format are reduced to standard matrix rules. If TT–ranks are reasonably small, then the TT decomposition has several suitable properties allowing a fast tensor element evaluation, effective storage and others [63, 64]. Note that TT–Cross method exploits the well–known matrix cross interpolation [31] algorithm. There are three main parameters defining the TT–docking algorithm performance, and their optimal values are determined during the SOL–P validation [93, 94]: the number of discretization points for each variable corresponding to each degree of freedom, the optimal value is  $2^{16}$ , the maximal TT–rank, the optimal rank is 4, and the number of iterations of the TT–algorithm, the optimal number is 15. SOL–P is written on C++ with usage of BLAS and LAPACK libraries and it uses the parallel MPI. Details of the TT–docking algorithm and the organization of performance of the SOL–P program are presented in [61, 93, 94].

SOL–P is successfully used for the docking flexible ligands into the proteins with moveable atoms. The results of such docking are described in [93, 94] where results of successful docking with 157 degrees of freedom are presented. The main distinction of this realization from the widely used ensemble docking is that the global energy minimum search is performed with variations of all variables describing ligand and protein degrees of freedom simultaneously and equally. Actually there are no special restrictions on the number of mobile protein atoms, but these atoms can move only in respective cubes with edges of about 1 Å and centered at the position of these atoms in the structure taken from PDB. Such investigations have shown that movements of some protein atoms in the docking process improves positioning accuracy of docking [93, 94]. The effectiveness of the TT–docking algorithm makes it possible to dock by SOL–P flexible ligands, oligopeptides, with a large number of torsions (more than 15 – 20) [87]. Such flexible ligands cannot be docked usually by classical docking programs.

### 2.6.3. Quasi–docking

Quasi–docking is a combination of a force field and quantum chemical methods [89]. The validation of the FLM generalized docking program and investigations with it the spectra of low energy minima of different test sets of protein–ligand complexes lead to the idea of the quasi–docking procedure. Quasi–docking has been used for the comparison of quality of docking with different force fields and quantum–chemical methods [88, 89]. Quasi–docking consists of two steps. At the first step sufficiently wide range of low energy minima of a protein–ligand complex is found with the help of the FLM docking program, the energy is being computed in the frame of the MMFF94 force field [32]. At the second step energies of all these minima

are recalculated with other force fields or quantum-chemical methods. It has been shown that when the protein-ligand energy is calculated with the CHARMM force field and the respective GBSW implicit solvent model ligand positioning accuracy is better than in the case of the energy calculated with the MMFF94 force field and with the respective PCM implicit solvent model [89]. On the other hand the docking accuracy with CHARMM is worse than with either PM6-D3H4X [101, 102], or PM7 [83] quantum-chemical methods used for the energy calculations with the COSMO implicit solvent model [43]. These quantum-chemical methods were developed rather recently comparing with older AM1 or PM3 methods, which had been widely used during latest 30 years. PM6-D3H4X and PM7 methods overcome the main limitations of older semiempirical methods describing well dispersion interactions and hydrogen bonds at the level of DFT *ab initio* methods [34]. These quantum-chemical semiempirical methods and the COSMO solvent model are implemented in the MOPAC (<http://OpenMOPAC.net>) package [84] where the module MOZYME gives an opportunity to calculate sufficiently quickly the whole protein-ligand complexes containing thousands of atoms. So, quasi-docking actually gives an opportunity to perform docking with quantum-chemical methods which demonstrate the best docking accuracy. A sufficiently wide low energy minima spectrum should be found at the first step of quasi-docking and its optimal width, when the energy is calculated with MMFF94 without solvent, is determined to perform minimal quantity of quantum-chemical recalculations [48]. Certainly, each step of the quasi-docking procedure essentially exploits supercomputing.

## Conclusion

Materials presented in this review show that supercomputers are used for docking more and more widely. To some extent this is amazing because the vast majority of docking programs are still focused on working on laptops and workstations. However, the needs to increase effectiveness of molecular modeling application to drug design, to perform docking more accurately and to screen millions of virtual and on-the-shelf compounds force to attract supercomputer resources for docking. The necessity to use large computing resources for docking appeared more than 10 years ago when grid technologies were used for virtual screening of large libraries of compounds using docking. Currently two directions of supercomputer docking are formed. First, the adaptation of existing docking programs to supercomputers to perform effectively docking of hundreds of thousands or millions of ligands. Two problems should be solved on this road: to reach high effectiveness of multi-core parallel calculations of a great amount of ligands, and to perform a fast and effective analysis of the very large arrays of results yielded by docking. Second direction of the supercomputer docking focuses on a more accurate docking of a single ligand. Along this road novel docking programs are created which are free from simplifications and approximations inherent in classical docking programs, and the novel programs can be used to obtain with maximal accuracy the spectrum of low energy minima of the protein-ligand system, which forms its configuration space and actually defines the free energy of the system. This road is much more tortuous and hard because one should overcome both computing and physical-chemical problems. The outlines of quantum-chemical docking is appearing ahead along this road. However, efforts to achieve this goal will be justified by attaining much higher efficiency in the use of docking programs for the drug development.

## Acknowledgments

The work was financially supported by the Russian Science Foundation, Agreement No. 15–11–00025-II. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University, including the Lomonosov supercomputer [100].

*This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.*

## References

1. Allen, W.J., Balius, T.E., Mukherjee, S., *et al.*: Dock 6: Impact of new features and current docking performance. *Journal of Computational Chemistry* 36(15), 1132–56 (2015), DOI: 10.1002/jcc.23905
2. Arcon, J.P., Defelipe, L.A., Modenutti, C.P., *et al.*: Molecular dynamics in mixed solvents reveals protein-ligand interactions, improves docking, and allows accurate binding free energy predictions. *Journal of Chemical Information and Modeling* 57(4), 846–863 (2017), DOI: 10.1021/acs.jcim.6b00678
3. Basciu, A., Mallocci, G., Pietrucci, F., *et al.*: Holo-like and druggable protein conformations from enhanced sampling of binding pocket volume and shape. *Journal of Chemical Information and Modeling* 59(4), 1515–1528 (2019), DOI: 10.1021/acs.jcim.8b00730
4. Beierlein, F., Lanig, H., Schrer, G., *et al.*: Quantum mechanical/molecular mechanical (QM/MM) docking: An evaluation for known test systems. *Molecular Physics – MOL PHYS* 101, 2469–2480 (2003), DOI: 10.1080/0026897031000092940
5. Bekker, G.J., Araki, M., Oshima, K., *et al.*: Dynamic docking of a medium-sized molecule to its receptor by multicanonical MD simulations. *The Journal of Physical Chemistry B* 123(11), 2479–2490 (2019), DOI: 10.1021/acs.jpcb.8b12419
6. Berman, H.M., Westbrook, J., Feng, Z., *et al.*: The protein data bank. *Nucleic Acids Research* 28(1), 235–42 (2000), DOI: 10.1093/nar/28.1.235
7. Best, R.B., Zhu, X., Shim, J., *et al.*: Optimization of the additive CHARMM all-atom protein force field targeting improved sampling of the backbone phi, psi and side-chain chi(1) and chi(2) dihedral angles. *Journal of Chemical Theory and Computation* 8(9), 3257–3273 (2012), DOI: 10.1021/ct300400x
8. Bikadi, Z., Hazai, E.: Application of the PM6 semi-empirical method to modeling proteins enhances docking accuracy of AutoDock. *Journal of cheminformatics* 1, 15–15 (2009), DOI: 10.1186/1758-2946-1-15
9. Bolcato, G., Cuzzolin, A., Bissaro, M., *et al.*: Can we still trust docking results? An extension of the applicability of DockBench on PDBbind database. *International journal of molecular sciences* 20(14), 3558 (2019), DOI: 10.3390/ijms20143558

10. ten Brink, T., Exner, T.E.: Influence of protonation, tautomeric, and stereoisomeric states on protein–ligand docking results. *Journal of Chemical Information and Modeling* 49(6), 1535–1546 (2009), DOI: 10.1021/ci800420z
11. ten Brink, T., Exner, T.E.: pka based protonation states and microspecies for protein–ligand docking. *Journal of Computer–Aided Molecular Design* 24(11), 935–942 (2010), DOI: 10.1007/s10822-010-9385-x
12. Brozell, S.R., Mukherjee, S., Balius, T.E., *et al.*: Evaluation of dock 6 as a pose generation and database enrichment tool. *Journal of Computer–Aided Molecular Design* 26(6), 749–773 (2012), DOI: 10.1007/s10822-012-9565-y
13. Burley, S.K., Berman, H.M., Bhikadiya, C., *et al.*: RCSB Protein Data Bank: biological macromolecular structures enabling research and education in fundamental biology, biomedicine, biotechnology and energy. *Nucleic Acids Research* 47(D1), D464–D474 (2018), DOI: 10.1093/nar/gky1004
14. Chen, Y.C.: Beware of docking! *Trends in Pharmacological Sciences* 36(2), 78–95 (2015), DOI: 10.1016/j.tips.2014.12.001
15. Chung, J.Y., Hah, J.M., Cho, A.E.: Correlation between performance of QM/MM docking and simple classification of binding sites. *Journal of Chemical Information and Modeling* 49(10), 2382–2387 (2009), DOI: 10.1021/ci900231p
16. Cole, D.J., Tirado-Rives, J., Jorgensen, W.L.: Molecular dynamics and monte carlo simulations for protein–ligand binding and inhibitor design. *Biochimica et biophysica acta* 1850(5), 966–971 (2015), DOI: 10.1016/j.bbagen.2014.08.018
17. Collignon, B., Schulz, R., Smith, J.C., *et al.*: Task–parallel message passing interface implementation of Autodock4 for docking of very large databases of compounds using high–performance super–computers. *Journal of Computational Chemistry* 32(6), 1202–9 (2011), DOI: 10.1002/jcc.21696
18. De Vivo, M., Cavalli, A.: Recent advances in dynamic docking for drug discovery. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 7(6), e1320 (2017), DOI: 10.1002/wcms.1320
19. Dixit, S.B., Chipot, C.: Can absolute free energies of association be estimated from molecular mechanical simulations? The biotin–streptavidin system revisited. *The Journal of Physical Chemistry A* 105(42), 9795–9799 (2001), DOI: 10.1021/jp011878v
20. Dolezal, R., Nepovimova, E., Melikova, M., *et al.*: Structure–based virtual screening for novel modulators of human orexin 2 receptor with cloud systems and supercomputers. In: *Advanced Topics in Intelligent Information and Database Systems*, pp. 161–171. Springer International Publishing (2017), DOI: 10.1007/978-3-319-56660-3\_15
21. Ellingson, S.R., Dakshanamurthy, S., Brown, M., *et al.*: Accelerating Virtual High–Throughput Ligand Docking: current technology and case study on a petascale supercomputer. *Concurrency and computation : practice & experience* 26(6), 1268–1277 (2014), DOI: 10.1002/cpe.3070

22. Elovely, K.M., Doerksen, R.J.: Docking challenge: protein sampling and molecular docking performance. *Journal of chemical information and modeling* 53(8), 1934–1945 (2013), DOI: 10.1021/ci400040d
23. Evangelista Falcon, W., Ellingson, S.R., Smith, J.C., *et al.*: Ensemble docking in drug discovery: How many protein configurations from molecular dynamics simulations are needed to reproduce known ligand binding? *The Journal of Physical Chemistry B* 123(25), 5189–5195 (2019), DOI: 10.1021/acs.jpcb.8b11491
24. Feixas, F., Lindert, S., Sinko, W., *et al.*: Exploring the role of receptor flexibility in structure-based drug discovery. *Biophysical chemistry* 186, 31–45 (2014), DOI: 10.1016/j.bpc.2013.10.007
25. Filamofitsky, M.P.: The system X–Com for metacomputing support: architecture and technology. *Numerical Methods and Programming* 5(2), 1–9 (in Russian) (2004)
26. Forli, S., Huey, R., Pique, M.E., *et al.*: Computational protein–ligand docking and virtual drug screening with the AutoDock suite. *Nature protocols* 11(5), 905–19 (2016), DOI: 10.1038/nprot.2016.051
27. Friesner, R.A., Murphy, R.B., Repasky, M.P., *et al.*: Extra precision glide: docking and scoring incorporating a model of hydrophobic enclosure for protein–ligand complexes. *Journal of Medicinal Chemistry* 49(21), 6177–96 (2006), DOI: 10.1021/jm051256o
28. Genheden, S., Nilsson, I., Ryde, U.: Binding affinities of factor Xa inhibitors estimated by thermodynamic integration and MM/GBSA. *Journal of Chemical Information and Modeling* 51(4), 947–958 (2011), DOI: 10.1021/ci100458f
29. Gibbs, N., Clarke, A.R., Sessions, R.B.: Ab initio protein structure prediction using physicochemical potentials and a simplified off-lattice model. *Proteins: Structure, Function, and Bioinformatics* 43(2), 186–202 (2001), DOI: 10.1002/1097-0134(20010501)43:2%3C186::aid-prot1030%3E3.0.co;2-1
30. Gioia, D., Bertazzo, M., Recanatini, M., *et al.*: Dynamic docking: A paradigm shift in computational drug discovery. *Molecules (Basel, Switzerland)* 22(11), 2029 (2017), DOI: 10.3390/molecules22112029
31. Goreinov, S., Tyrtyshnikov, E.: The maximal–volume concept in approximation by low–rank matrices. *Contemporary Mathematics* 268, 47–51 (2001)
32. Halgren, T.A.: Merck molecular force field. I. Basis, form, scope, parameterization, and performance of MMFF94. *Journal of Computational Chemistry* 17(5-6), 490–519 (1996), DOI: 10.1002/(SICI)1096-987X(199604)17:5/6%3C490::AID-JCC1%3E3.0.CO;2-P
33. Honegr, J., Dolezal, R., Malinak, D., *et al.*: Rational design of a new class of toll-like receptor 4 (TLR4) tryptamine related agonists by means of the structure– and ligand–based virtual screening for vaccine adjuvant discovery. *Molecules* 23(1) 102 (2018), DOI: 10.3390/molecules23010102
34. Hostaš, J., Řezáč, J., Hobza, P.: On the performance of the semiempirical quantum mechanical PM6 and PM7 methods for noncovalent interactions. *Chemical Physics Letters* 568–569(Supplement C), 161–166 (2013), DOI: 10.1016/j.cplett.2013.02.069

35. Huang, N., Shoichet, B.K., Irwin, J.J.: Benchmarking sets for molecular docking. *Journal of Medicinal Chemistry* 49(23), 6789–6801 (2006), DOI: 10.1021/jm0608356
36. Irwin, J.J., Sterling, T., Mysinger, M.M., *et al.*: ZINC: A free tool to discover chemistry for biology. *Journal of Chemical Information and Modeling* 52(7), 1757–1768 (2012), DOI: 10.1021/ci3001277
37. Jacq, N., Breton, V., Chen, H.Y., *et al.*: Virtual screening on large scale grids. *Parallel Computing* 33(4), 289–301 (2007), DOI: 10.1016/j.parco.2007.02.010
38. Jaghoori, M.M., Bleijlevens, B., Olabarriaga, S.D.: 1001 Ways to run AutoDock Vina for virtual screening. *Journal of Computer-Aided Molecular Design* 30(3), 237–249 (2016), DOI: 10.1007/s10822-016-9900-9
39. Jorgensen, W.L., Tirado-Rives, J.: Potential energy functions for atomic-level simulations of water and organic and biomolecular systems. *Proceedings of the National Academy of Sciences of the United States of America* 102(19), 6665–70 (2005), DOI: 10.1073/pnas.0408037102
40. Kalliokoski, T., Salo, H.S., Lahtela-Kakkonen, M., *et al.*: The effect of ligand-based tautomer and protomer prediction on structure-based virtual screening. *Journal of Chemical Information and Modeling* 49(12), 2742–2748 (2009), DOI: 10.1021/ci900364w
41. Kantardjieff, A.A.: Quantum.ligand.dock: protein-ligand docking with quantum entanglement refinement on a GPU system. *Nucleic acids research* 40(Web Server issue), W415–W422 (2012), DOI: 10.1093/nar/gks515
42. Khavrutskii, I.V., Wallqvist, A.: Improved binding free energy predictions from single-reference thermodynamic integration augmented with hamiltonian replica exchange. *Journal of Chemical Theory and Computation* 7(9), 3001–3011 (2011), DOI: 10.1021/ct2003786
43. Klamt, A., Schuurmann, G.: COSMO: a new approach to dielectric screening in solvents with explicit expressions for the screening energy and its gradient. *Journal of the Chemical Society, Perkin Transactions 2* (5), 799–805 (1993), DOI: 10.1039/P29930000799
44. Klebe, G.: The use of thermodynamic and kinetic data in drug discovery: Decisive insight or increasing the puzzlement? *ChemMedChem* 10(2), 229–231 (2015), DOI: 10.1002/cmdc.201402521
45. Klimovich, P.V., Shirts, M.R., Mobley, D.L.: Guidelines for the analysis of free energy calculations. *Journal of computer-aided molecular design* 29(5), 397–411 (2015), DOI: 10.1007/s10822-015-9840-9
46. Kollman, P.: Free energy calculations: Applications to chemical and biochemical phenomena. *Chemical Reviews* 93(7), 2395–2417 (1993), DOI: 10.1021/cr00023a004
47. Kutov, D.K., Katkova, E.V., Sulimov, E.V., *et al.*: Influence of the method of hydrogen atoms incorporation into the target protein on the protein-ligand binding energy. *Bulletin of the South Ural State University, Ser. Mathematical Modelling, Programming & Computer Software* 10(3), 94–107 (2017), DOI: 10.14529/mmp170308

48. Kutov, D.C., Sulimov, A.V., Sulimov, V.B.: Supercomputer docking: Investigation of low energy minima of protein–ligand complexes. *Supercomputing Frontiers and Innovations* 5(3), 134–137 (2018), DOI: 10.14529/jsfi180326
49. Lamm Ribeiro, J.a.M., Tiwary, P.: Toward achieving efficient and accurate ligand–protein unbinding with deep learning and molecular dynamics through RAVE. *Journal of Chemical Theory and Computation* 15(1), 708–719 (2019), DOI: 10.1021/acs.jctc.8b00869
50. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. *Nature* 521, 436 (2015), DOI: 10.1038/nature14539
51. Lushchekina, S.V., Makhaeva, G.F., Novichkova, D.A., *et al.*: Supercomputer modeling of dual–site acetylcholinesterase (AChE) inhibition. *Supercomputing Frontiers and Innovations* 5(4), 89–97 (2018), DOI: 10.14529/jsfi180410
52. McIntosh-Smith, S., Price, J., Sessions, R.B., *et al.*: High performance in silico virtual drug screening on many–core processors. *The International Journal of High Performance Computing Applications* 29(2), 119–134 (2015), DOI: 10.1177/1094342014528252
53. Michel, J., Foloppe, N., Essex, J.W.: Rigorous free energy calculations in structure–based drug design. *Molecular Informatics* 29(8-9), 570–578 (2010), DOI: 10.1002/minf.201000051
54. Mobley, D.L., Gilson, M.K.: Predicting binding free energies: Frontiers and benchmarks. *Annual Review of Biophysics* 46(1), 531–558 (2017), DOI: 10.1146/annurev-biophys-070816-033654
55. Morris, G.M., Huey, R., Lindstrom, W., *et al.*: Autodock4 and AutoDockTools4: Automated docking with selective receptor flexibility. *Journal of Computational Chemistry* 30(16), 2785–91 (2009), DOI: 10.1002/jcc.21256
56. Morris, G.M., Goodsell, D.S., Halliday, R.S., *et al.*: Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 19(14), 1639–1662 (1998), DOI: 10.1002/(SICI)1096-987X(19981115)19:14;1639::AID-JCC10;3.0.CO;2-B
57. Neves, M.A., Totrov, M., Abagyan, R.: Docking and scoring with ICM: the benchmarking results and strategies for improvement. *Journal of Computer–Aided Molecular Design* 26(6), 675–686 (2012), DOI: 10.1007/s10822-012-9547-0
58. Nguyen, D.D., Wei, G.W.: Agl–score: Algebraic graph learning score for proteinligand binding scoring, ranking, docking, and screening. *Journal of Chemical Information and Modeling* 59(7), 3291–3304 (2019), DOI: 10.1021/acs.jcim.9b00334
59. Nogueira, M.S., Koch, O.: The development of target–specific machine learning models as scoring functions for docking–based target prediction. *Journal of Chemical Information and Modeling* 59(3), 1238–1252 (2019), DOI: 10.1021/acs.jcim.8b00773
60. Oferkin, I.V., Katkova, E.V., Sulimov, A.V., *et al.*: Evaluation of docking target functions by the comprehensive investigation of protein–ligand energy minima. *Advances in Bioinformatics* 2015, 126858 (2015), DOI: 10.1155/2015/126858

61. Oferkin, I.V., Zheltkov, D.A., Tyrtyshnikov, E.E., *et al.*: Evaluation of the docking algorithm based on Tensor Train global optimization. Bulletin of the South Ural State University, Ser. Mathematical Modelling, Programming & Computer Software 8(4), 83–99 (2015), DOI: 10.14529/mmp150407
62. Ohue, M., Shimoda, T., Suzuki, S., *et al.*: Megadock 4.0: an ultra-high-performance protein–protein docking software for heterogeneous supercomputers. Bioinformatics (Oxford, England) 30(22), 3281–3283 (2014), DOI: 10.1093/bioinformatics/btu532
63. Oseledets, I.: Tensor–train decomposition. SIAM Journal on Scientific Computing 33(5), 2295–2317 (2011), DOI: 10.1137/090752286
64. Oseledets, I., Tyrtyshnikov, E.: Breaking the curse of dimensionality, or how to use SVD in many dimensions. SIAM Journal on Scientific Computing 31(5), 3744–3759 (2009), DOI: 10.1137/090748330
65. Oseledets, I., Tyrtyshnikov, E.: TT–cross approximation for multidimensional arrays. Linear Algebra and its Applications 432(1), 70–88 (2010), DOI: 10.1016/j.laa.2009.07.024
66. Pagadala, N.S., Syed, K., Tuszyński, J.: Software for molecular docking: a review. Biophysical Reviews 9(2), 91–102 (2017), DOI: 10.1007/s12551-016-0247-1
67. Pan, A.C., Borhani, D.W., Dror, R.O., *et al.*: Molecular determinants of drugreceptor binding kinetics. Drug Discovery Today 18(13), 667–673 (2013), DOI: 10.1016/j.drudis.2013.02.007
68. Park, M.S., Gao, C., Stern, H.A.: Estimating binding affinities by docking/scoring methods using variable protonation states. Proteins: Structure, Function, and Bioinformatics 79(1), 304–314 (2011), DOI: 10.1002/prot.22883
69. Pei, J., Zheng, Z., Kim, H., *et al.*: Random forest refinement of pairwise potentials for proteinligand decoy detection. Journal of Chemical Information and Modeling 59(7), 3305–3315 (2019), DOI: 10.1021/acs.jcim.9b00356
70. Perthold, J.W., Oostenbrink, C.: Accelerated enveloping distribution sampling: Enabling sampling of multiple end states while preserving local energy minima. The Journal of Physical Chemistry B 122(19), 5030–5037 (2018), DOI: 10.1021/acs.jpcb.8b02725
71. Ragoza, M., Hochuli, J., Idrobo, E., *et al.*: Proteinligand scoring with convolutional neural networks. Journal of Chemical Information and Modeling 57(4), 942–957 (2017), DOI: 10.1021/acs.jcim.6b00740
72. Rarey, M., Kramer, B., Lengauer, T., *et al.*: A fast flexible docking method using an incremental construction algorithm. Journal of Molecular Biology 261(3), 470–489 (1996), DOI: 10.1006/jmbi.1996.0477
73. Rekapalli, B., Vose, A., Giblock, P.: HSPp–BLAST: Highly scalable parallel PSI–BLAST for very large–scale sequence searches. In: 4th International Conference on Bioinformatics and Computational Biology 2012, BICoB 2012, 12–14 March 2012, Las Vegas, Nevada, USA. pp. 37–42 (03 2012)

74. Ribeiro, J.M.L., Bravo, P., Wang, Y., *et al.*: Reweighted autoencoded variational Bayes for enhanced sampling (RAVE). *The Journal of Chemical Physics* 149(7), 072301 (2018), DOI: 10.1063/1.5025487
75. Riniker, S., Christ, C.D., Hansen, N., *et al.*: Comparison of enveloping distribution sampling and thermodynamic integration to calculate binding free energies of phenylethanolamine N-methyltransferase inhibitors. *The Journal of Chemical Physics* 135(2), 024105 (2011), DOI: 10.1063/1.3604534
76. Romanov, A.N., Jabin, S.N., Martynov, Y.B., *et al.*: Surface generalized born method: A simple, fast, and precise implicit solvent model beyond the coulomb approximation. *The Journal of Physical Chemistry. A, Molecules, spectroscopy, kinetics, environment & general theory* 108(43), 9323–9327 (2004), DOI: 10.1021/jp046721s
77. Ryde, U., Sderhjelm, P.: Ligand-binding affinity estimates supported by quantum-mechanical methods. *Chemical Reviews* 116(9), 5520–5566 (2016), DOI: 10.1021/acs.chemrev.5b00630
78. Sadovnichii, V.A., Sulimov, V.B.: Supercomputing technologies in medicine. In: *Supercomputing Technologies in Science*, pp. 16–23. Moscow University Publishing (2009)
79. Sinauridze, E.I., Romanov, A.N., Gribkova, I.V., *et al.*: New synthetic thrombin inhibitors: molecular design and experimental verification. *PLoS One* 6(5), e19969 (2011), DOI: 10.1371/journal.pone.0019969
80. Sliwoski, G., Kothiwale, S., Meiler, J., *et al.*: Computational methods in drug discovery. *Pharmacological Reviews* 66(1), 334–395 (2013), DOI: 10.1124/pr.112.007336
81. Sobolev, S.I.: Effective performance of distributed computing environments, pp. 249–258 (in Russian). Moscow State University Press (2008)
82. Spitaleri, A., Decherchi, S., Cavalli, A., *et al.*: Fast dynamic docking guided by adaptive electrostatic bias: The MD–binding approach. *Journal of Chemical Theory and Computation* 14(3), 1727–1736 (2018), DOI: 10.1021/acs.jctc.7b01088
83. Stewart, J.J.: Optimization of parameters for semiempirical methods VI: more modifications to the NDDO approximations and re-optimization of parameters. *Journal of Molecular Modeling* 19(1), 1–32 (2013), DOI: 10.1007/s00894-012-1667-x
84. Stewart, J.J.P.: Mopac2016. <http://OpenMOPAC.net> (2016)
85. Still, W.C., Tempczyk, A., Hawley, R.C., *et al.*: Semianalytical treatment of solvation for molecular mechanics and dynamics. *Journal of the American Chemical Society* 112(16), 6127–6129 (1990), DOI: 10.1021/ja00172a038
86. Strecker, C., Meyer, B.: Plasticity of the binding site of renin: Optimized selection of protein structures for ensemble docking. *Journal of Chemical Information and Modeling* 58(5), 1121–1131 (2018), DOI: 10.1021/acs.jcim.8b00010

87. Sulimov, A., Kutov, D., Ilin, I., *et al.*: Supercomputer docking with a large number of degrees of freedom. In: Devillers, E.J., Geronikaki, A. (eds.) 10th International Symposium on Computational Methods in Toxicology and Pharmacology Integrating Internet Resources 2019, CMPTI 2019, 23-27 June 2019, Ioannina, Greece. p. 24 (2019)
88. Sulimov, A.V., Kutov, D.C., Katkova, E.V., *et al.*: Combined docking with classical force field and quantum chemical semiempirical method PM7. Advances in Bioinformatics 2017, 7167691 (2017), DOI: 10.1155/2017/7167691
89. Sulimov, A.V., Kutov, D.C., Katkova, E.V., *et al.*: New generation of docking programs: Supercomputer validation of force fields and quantum-chemical methods for docking. Journal of Molecular Graphics and Modelling 78, 139–147 (2017), DOI: 10.1016/j.jmgm.2017.10.007
90. Sulimov, A.V., Kutov, D.C., Katkova, E.V., *et al.*: Search for approaches to improving the calculation accuracy of the proteinligand binding energy by docking. Russian Chemical Bulletin, International Edition 66(10), 1913–1924 (2017), DOI: 10.1007/s11172-017-1966-6
91. Sulimov, A.V., Kutov, D.C., Oferkin, I.V., *et al.*: Application of the docking program SOL for CSAR benchmark. Journal of Chemical Information and Modeling 53(8), 1946–56 (2013), DOI: 10.1021/ci400094h
92. Sulimov, A.V., Kutov, D.C., Sulimov, V.B.: Parallel supercomputer docking program of the new generation: Finding low energy minima spectrum. In: 4th Russian Supercomputing Days 2018, RuSCDays 2018, 2425 September 2018, Moscow, Russia. vol. 965, pp. 314–330 (2018), DOI: 10.1007/978-3-030-05807-4\_27
93. Sulimov, A.V., Zheltkov, D.A., Oferkin, I.V., *et al.*: Evaluation of the novel algorithm of flexible ligand docking with moveable target–protein atoms. Computational and Structural Biotechnology Journal 15, 275–285 (2017), DOI: 10.1016/j.csbj.2017.02.004
94. Sulimov, A.V., Zheltkov, D.A., Oferkin, I.V., *et al.*: Tensor train global optimization: Application to docking in the configuration space with a large number of dimensions. In: 3rd Russian Supercomputing Days. vol. 793, pp. 151–167 (2017), DOI: 10.1007/978-3-319-71255-0\_12
95. Sulimov, V.B., Sulimov, A.: Docking: molecular modeling for drug discovery. AINTELL, Moscow (2017)
96. Sulimov, V.B., Kutov, D.C., Sulimov, A.V.: Advances in docking. Current Medicinal Chemistry 26(37), 1–25 (2019), DOI: 10.2174/0929867325666180904115000
97. Trager, R.E., Giblock, P., Soltani, S., *et al.*: Docking optimization, variance and promiscuity for large-scale drug-like chemical space using high performance computing architectures. Drug Discovery Today 21(10), 1672–1680 (2016), DOI: 10.1016/j.drudis.2016.06.023
98. Trott, O., Olson, A.J.: Autodock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. Journal of Computational Chemistry 31(2), 455–61 (2010), DOI: 10.1002/jcc.21334

99. Ustach, V.D., Lakkaraju, S.K., Jo, S., *et al.*: Optimization and evaluation of site-identification by ligand competitive saturation (SILCS) as a tool for target-based ligand optimization. *Journal of Chemical Information and Modeling* 59(6), 3018–3035 (2019), DOI: 10.1021/acs.jcim.9b00210
100. Voevodin, V.V., Antonov, A.S., Nikitenko, D.A., *et al.*: Supercomputer lomonosov-2: Large scale, deep monitoring and fine analytics for the user community. *Supercomputing Frontiers and Innovations* 6(2), 4–11 (2019), DOI: 10.14529/jsfi190201
101. Řezáč, J., Hobza, P.: A halogen–bonding correction for the semiempirical PM6 method. *Chemical Physics Letters* 506(4), 286–289 (2011), DOI: 10.1016/j.cplett.2011.03.009
102. Řezáč, J., Hobza, P.: Advanced corrections of hydrogen bonding and dispersion for semiempirical quantum mechanical methods. *Journal of Chemical Theory and Computation* 8(1), 141–151 (2012), DOI: 10.1021/ct200751e
103. Wang, J.C., Lin, J.H., Chen, C.M., *et al.*: Robust scoring functions for protein–ligand interactions with quantum chemical charge models. *Journal of chemical information and modeling* 51(10), 2528–2537 (2011), DOI: 10.1021/ci200220v
104. Wang, J., Wolf, R.M., Caldwell, J.W., *et al.*: Development and testing of a general amber force field. *Journal of Computational Chemistry* 25(9), 1157–1174 (2004), DOI: 10.1002/jcc.20035
105. Wang, L., Wu, Y., Deng, Y., *et al.*: Accurate and reliable prediction of relative ligand binding potency in prospective drug discovery by way of a modern free–energy calculation protocol and force field. *Journal of the American Chemical Society* 137(7), 2695–2703 (2015), DOI: 10.1021/ja512751q
106. Xie, B., Clark, J.D., Minh, D.D.L.: Efficiency of stratification for ensemble docking using reduced ensembles. *Journal of Chemical Information and Modeling* 58(9), 1915–1925 (2018), DOI: 10.1021/acs.jcim.8b00314
107. Yuriev, E., Holien, J., Ramsland, P.A.: Improvements, trends, and new ideas in molecular docking: 20122013 in review. *Journal of Molecular Recognition* 28(10), 581604 (2015), DOI: 10.1002/jmr.2471
108. Zhang, X., Wong, S.E., Lightstone, F.C.: Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines. *Journal of Computational Chemistry* 34(11), 915–927 (2013), DOI: 10.1002/jcc.23214
109. Zheltkov, D.A., Oferkin, I.V., Katkova, E.V., *et al.*: TT-Dock: a docking method based on tensor train decompositions. *Numerical Methods and Programming* 14(3), 279–291 (in Russian) (2013)
110. van Zundert, G., Rodrigues, J., Trellet, M., *et al.*: The HADDOCK2.2 Web Server: User-friendly integrative modeling of biomolecular complexes. *Journal of Molecular Biology* 428(4), 720–725 (2016), DOI: 10.1016/j.jmb.2015.09.014

# Automatic Port to OpenACC/OpenMP for Physical Parameterization in Climate and Weather Code Using the CLAW Compiler

**Valentin Clement<sup>1</sup>** , **Philippe Marti<sup>1</sup>** , **Xavier Lapillonne<sup>2</sup>** ,  
**Oliver Fuhrer<sup>2</sup>** , **William Sawyer<sup>3</sup>** 

© The Authors 2019. This paper is published with open access at SuperFri.org

In order to benefit from emerging high-performance computing systems, weather and climate models need to be adapted to run efficiently on different hardware architectures such as accelerators. This is a major challenge for existing community models that represent extremely large codebase written in Fortran. Large parts of the code can be ported using OpenACC compiler directives but for time-critical components such as physical parameterizations, code restructuring and optimizations specific to a hardware architecture are necessary to obtain high performance. In an effort to retain a single source code for multiple target architectures, the CLAW Compiler and the CLAW Single Column Abstraction were introduced. We report on the extension of the CLAW SCA to handle ELEMENTAL functions and subroutines. We demonstrate the new capability on the JSBACH land surface scheme of the ICON climate model. With the extension, JSBACH can be automatically ported to OpenACC or OpenMP for accelerators with minimal to no change to the original code.

*Keywords:* compiler, directive, GPU, OpenACC, OpenMP, automatic port.

## Introduction

Numerical Weather Prediction and Climate modeling can highly benefit from computer technologies advances by increasing resolution or model complexity. In the recent years, architectures such as Graphics Processing Unit (GPU) have emerged for scientific high performance computing offering new opportunities. Most of the current Numerical Weather Prediction and Climate models are large Fortran based community codes which require to be adapted or re-written to run on non traditional CPU architectures such as GPU. In order to prepare for heterogenous supercomputer architectures, the global weather and climate model ICON [2, 4] is being ported to accelerators. The major part of the porting is currently achieved using OpenACC [10] compiler directives. For many code sections simple insertion of OpenACC compiler directives is an appropriate porting approach. But, for time-critical components such as the physical parameterizations, code restructuring and optimizations it is necessary to obtain optimal performance [6].

In some cases, the required code restructuring and optimization for GPU architectures have a negative impact when running the same code on a CPU architecture. Multiple solutions to address this problem have been proposed to achieve performance portability across architectures [3, 5, 8]. Here we focus on CLAW [1], an open-source source-to-source translator that allows to perform architecture-specific code transformations with minimal code modifications.

The CLAW SCA is designed to address the physical parameterizations of atmospheric models in Fortran. Physical parameterizations are typically horizontally independent so each vertical column can be computed separately. With the CLAW SCA, the physical parameterizations are written in Fortran only considering the vertical dependencies while the horizontal directions are abstracted out. The CLAW Compiler can transform the code for a specific target architecture

<sup>1</sup>Centre for Climate System Modeling, Zurich, Switzerland

<sup>2</sup>Federal Office of Meteorology and Climatology MeteoSwiss, Zurich, Switzerland

<sup>3</sup>CSCS Swiss National Supercomputing Centre, Lugano, Switzerland

and insert horizontal loops and compiler directives such as OpenMP [11] for accelerator (version  $\geq 4.5$ ) or OpenACC.

In this paper, we focus on an CLAW SCA incorporation that extends CLAW capability to address such performance portability issues without imposing disturbing changes to the code base. We introduce a special case of the CLAW SCA connected with ELEMENTAL functions and subroutines. We demonstrate the new feature on the JSBACH land surface scheme [7] used in the ICON climate model. This paper is structured as follows: Section 1 describes the JSBACH land surface scheme and its use of ELEMENTALs. In Sections 2 and 3 we present the CLAW Compiler and the extension of SCA for ELEMENTALs. In Sections 4 and 5 we discuss our performance results and some code metrics. Finally, in Section 5 we draw our conclusion and discuss future work.

## 1. The JSBACH Land Surface Scheme and ELEMENTAL

In this section, we briefly describe the concept of ELEMENTAL subroutines and functions and its application in the JSBACH land surface scheme. Further, we summarize its structure.

### 1.1. ELEMENTAL subroutines and functions

In Fortran, an ELEMENTAL function or subroutine is defined as a scalar operator. Dummy arguments as well as potential return value must be scalars but it may be called with arrays of arbitrary dimensionality as actual arguments. In this case, the operations defined in the function or the subroutine are applied element-wise on the full arrays. The main benefit of ELEMENTAL functions or subroutines is that it allows the user to write more compact code and allows the compiler to parallelize function or subroutine execution.

```

1 PROGRAM main
2 IMPLICIT NONE
3 INTEGER :: x, y, z
4 INTEGER, DIMENSION(10) :: xa, ya, za
5 x = 2; y = 10
6 xa(:) = 2; ya(:) = 5
7
8 ! Call ELEMENTAL with scalars
9 z = power(x, y)
10 print*, 'x ** y = ', z
11
12 ! Call ELEMENTAL with arrays
13 za(:) = power(xa(:), ya(:))
14 print*, 'xa(:) ** ya(:) = ', za
15
16 CONTAINS
17
18 ELEMENTAL FUNCTION power(a, b) RESULT(c)
19     INTEGER, INTENT(IN) :: a, b
20     INTEGER :: c
21     c = a ** b
22 END FUNCTION power
23 END PROGRAM

```

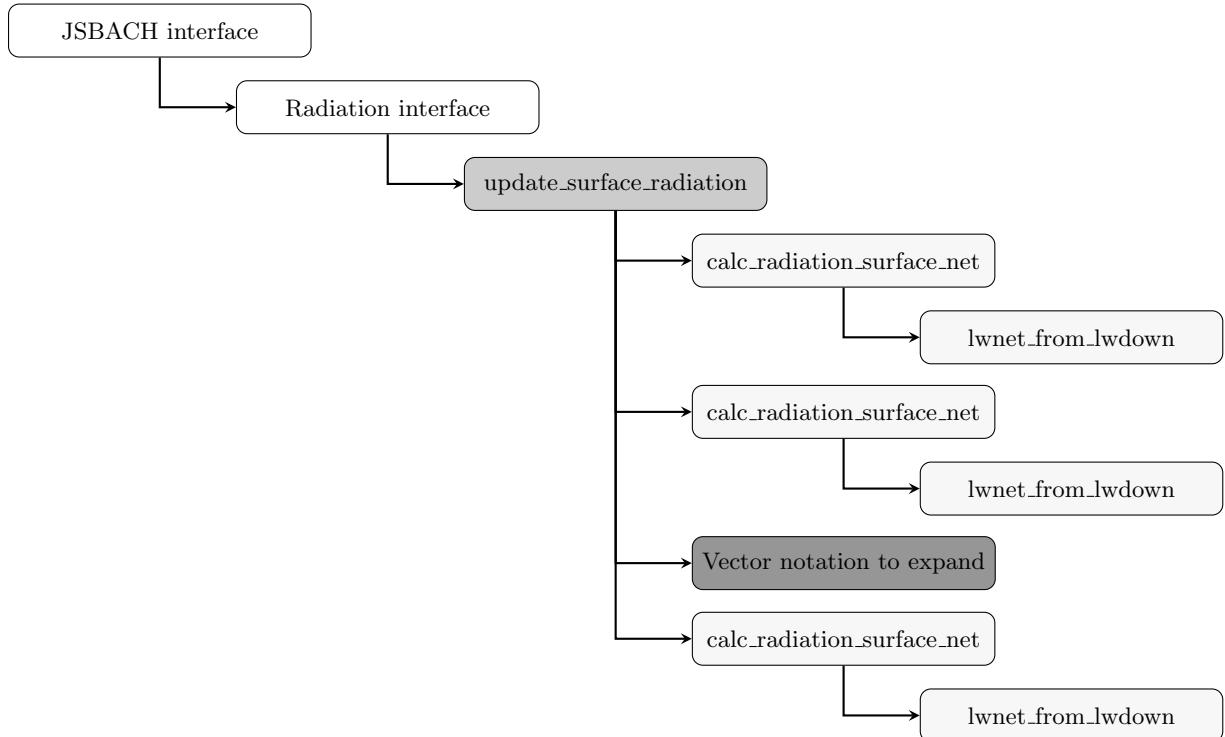
**Figure 1.** Code example with a basic ELEMENTAL function

A very basic ELEMENTAL function is implemented in Fig. 1 from line 18 to 22. First it is mentioned with scalar arguments at line 9, second, with arrays arguments at line 13.

## 1.2. JSBACH

JSBACH is the land surface scheme of the global ICON model in climate mode. It is designed to serve as a land surface boundary for the atmosphere in the coupled simulation but it can also be used offline as a standalone model. JSBACH simulates photosynthesis, phenology and land physics with hydrological and bio-geochemical cycles.

JSBACH code is written in 2008 Fortran object-oriented style. It is designed as a pipeline of tasks where each task is applied on various tiles of the soil depending on their properties. Each task retrieves a number of fields from the internal memory object and applies several ELEMENTAL functions and/or subroutines as well as vector operations to them.



**Figure 2.** Example of a typical task workflow using ELEMENTALS in JSBACH - update radiation surface from the radiation interface

To hide some aspects of the design such as the memory access and pointers, JSBACH includes a non-Fortran Domain Specific Language (DSL). The code must be preprocessed by a Python script in order to obtain standard Fortran code. Section 2.1 describes where this preprocessing sits in the CLAW Compiler workflow. The choices of using ELEMENTAL functions and subroutines and the JSBACH DSL significantly simplify the code for the domain scientist.

In its original form, the code is not suited to be ported easily to accelerators using OpenACC or OpenMP as compilers do not allow directives in PURE or ELEMENTAL functions and subroutines. In some cases vector notation can be handled by compilers with the `!$acc kernels` construct. But in practice, we have experienced several cases where the compiler was not able to determine that the kernel can be run in parallel and thus generated a sequential version of the code resulting in a significantly slower GPU execution comparing to the CPU version. This was especially true with older version of OpenACC compatible compiler but can be solved having reasonable amount of vector notation in an `!$acc kernels` block.

In order to execute the code on a GPU, we need to either fundamentally refactor JSBACH or to take advantage of a source-to-source translator such as CLAW. CLAW can automatize the port while taking advantage of the current information we can extract from it. The latter solution is the one described in this paper and has the advantage to bring portability across compiler directives by supporting OpenACC and OpenMP simultaneously.

Figure 2 is a typical call graph of a single task defined in JSBACH. `update_radiation_surface` is part of the radiation interface. This task is calling a single ELEMENTAL subroutine three times with different fields as arguments. This ELEMENTAL subroutine (`calc_radiation_surface_net`) calls another ELEMENTAL function.

As all ELEMENTAL code needs to be transformed to accept compiler directives, everything in the task is transformed to be executed on the accelerator.

**Table 1.** JSBACH tasks with the number of automatically generated kernel and the number of 1 dimensional (1D) and 2 dimensional (2D) input/output fields for each task

Interface	Task name	Nb. kernels	1D in	1D out	2D in	2D out
Radiation	surface_radiation	4	11	7	0	10
	radiation_par	5	10	6	4	4
	albedo	12	38	18	0	0
Phenology	phenology_logrop	11	30	21	0	0
	fpc	3	2	1	0	10
Hydrology	snow_and_skin_fraction	11	11	4	0	0
	soil_properties	2	7	0	7	7
	evaporation	3	16	6	0	0
	surface_hydrology	1	4	0	4	0
	soil_hydrology	3	3	3	1	0
	canopy_cond_unstressed	1	2	1	0	0
	canopy_cond_stressed	1	7	5	5	0
	water_balance	5	7	3	2	3
Surface energy balance	surface_energy_lake	7	16	16	0	0
	surface_energy_land	18	23	14	0	0
	surface_fluxes_lake	1	11	6	0	0
	surface_fluxes_land	1	7	4	0	0
	asselin_land	3	3	2	0	0
	snowmelt_correction	4	3	1	4	0
Soil snow energy	soil_and_snow_properties	14	2	4	2	2
	soil_and_snow_temperature	14	3	5	4	10
Assimilation	assimilation_scaling_factors	2	1	0	0	1
	canopy_cond_unstressed	2	4	1	3	1
Turbulence	humidity_scaling	10	14	3	3	0
	roughness	3	3	1	0	0

The full JSBACH model in its ICON configuration is composed of about 30 different tasks and represents roughly 30'000 lines of code. There is a total of 70 ELEMENTAL subroutines and functions to transform. Our approach is applied to all required tasks for a climate simulation.

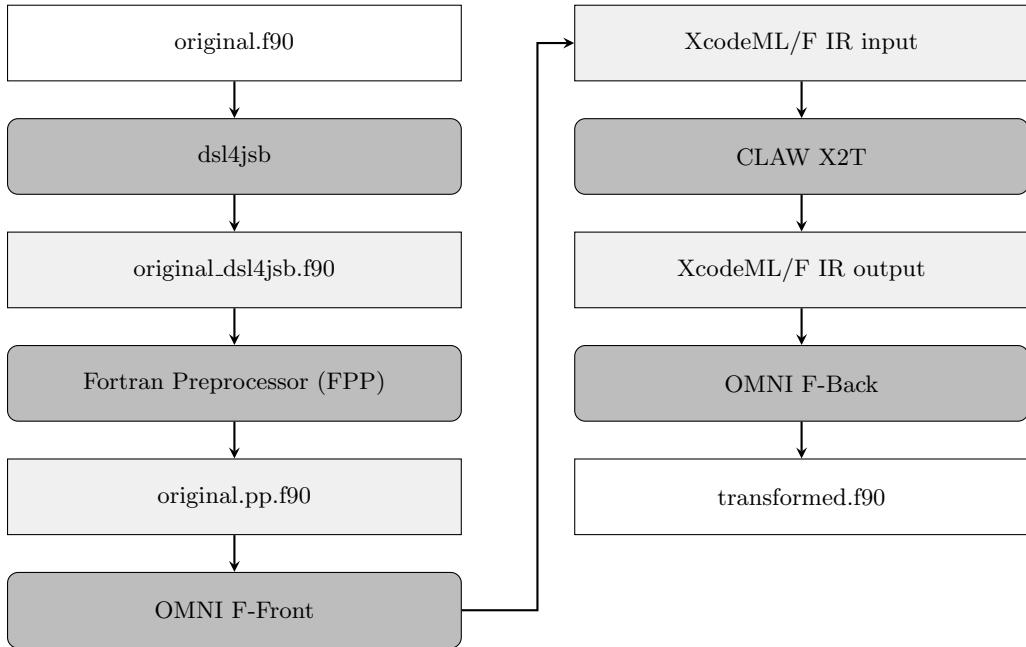
Table 1 describes the list of tasks automatically ported to GPU. For each tasks, the number of kernels generated and their corresponding inputs and outputs are detailed. 1 dimensional fields includes only the horizontal dimension where 2 dimensional include the number of soil layer as their second dimension.

## 2. The CLAW Compiler and the Single Column Abstraction

This section presents the CLAW Compiler and the Single Column Abstraction on which our work is based.

### 2.1. The Compiler

The CLAW Compiler is an extensible open-source source-to-source compiler for modern Fortran code. It is based on the OMNI Compiler Project [9].



**Figure 3.** CLAW Compiler workflow including the JSBACH DSL preprocessing

Figure 3 illustrates the internal workflow of the compiler. JSBACH code is pre-processed by a dedicated Python script which initially translates the JSBACH DSL to standard Fortran before entering the CLAW workflow. Fortran code is pre-processed and then parsed to the XcodeML/F Intermediate Representation (IR) [12]. This IR - represented as an Abstract Syntax Tree (AST) - is then manipulated by CLAW XcodeML to XcodeML Translator (X2T) to produce the refactored version of the code for a specific target with inserted directives. Finally, the IR is decompiled to standard Fortran code before being compiled by default compilers. Major contribution to extend its transformations for ELEMENTALS was introduced in the CLAW X2T.

### 2.2. CLAW Single Column Abstraction (SCA)

In weather and climate models, the physical parameterizations are, in most cases, single column processes. Each column of the three-dimensional computational domain can be computed

independently and does not have any data dependencies on its neighbors. The CLAW Single Column Abstraction DSL [1] was introduced to exploit this information. A CLAW SCA source code is fully standard Fortran code with the addition of specific CLAW directives. The code has no notion of the horizontal dimensions in the declaration of the fields as well as in the execution part. Loops iterating over horizontal dimension are omitted.

```
1 $ clawfc --target=gpu --directive=acc -o transformed.f90 original.f90
```

**Figure 4.** Invocation of the CLAW Compiler with a specified target architecture and desired directives

The SCA code is processed by the CLAW Compiler as shown in Fig. 4. The user defines the target architecture as well as the compiler directives to be used. Under the hood, the CLAW Compiler performs create a data dependency graph to decide where to create kernels and therefore which temporary fields have to be promoted. Based on this analysis it inserts the necessary iterations over the parallel dimensions and promotes the appropriate fields within the physical parameterization. In addition, the compiler inserts the required compiler directives to parallelize the loops and to manage data movement between the host and the device memory. The SCA transformation comes with various user-configurable options to manage promotion and data movement strategy, giving the end-user freedom to test various configurations.

### 3. Extension for ELEMENTAL Subroutines and Functions

An ELEMENTAL function or subroutine can be seen as a specific case of the CLAW SCA [1]. The land surface scheme is a point-wise computation on the grid that can be viewed as a column-wise computation with a limited number of vertical level.

The depth of the subroutine or function in the call graph determines the transformation applied to it. In Fig. 2, the leaf function `lwnet_from_lwdown` can be considered as a function to be inlined but `calc_radiation_surface_net` is a perfect candidate to apply SCA transformation rules.

Figure 5 is the original code of `calc_radiation_surface.net` processed by the user with CLAW SCA directive. Line 6 and 10 are defining the block of field coming from the model. These fields are often defined globally and therefore have to comply with the memory layout imposed by the calling model. The compiler then manages whether to apply promotion or not. The block directive also marks the whole subroutine as a SCA subroutine and thus activates the CLAW SCA transformation on it.

When a field is promoted or when a parallel iteration is inserted back into the code by the compiler, the dimension information is taken from the SCA model configuration file. This TOML formatted file defines the dimensions omitted in ELEMENTAL as well as layouts to be applied during the promotion transformation. This allows the user to investigate different data layouts depending on the target architecture. The user can update the default layout in the configuration file (Fig. 6) or add a layout clause to the `model-data` directive with the desired layout. The SCA model configuration file used for JSBACH is shown in Fig. 6. This file is read by the compiler before applying transformation.

The CLAW SCA transformation applied to non-ELEMENTAL subroutine or function will modifies the source code for both CPU and GPU targets. Since the original code with ELEMENTALS is already suited for CPU target, the extension of the SCA has no effect for this

```

1 ELEMENTAL SUBROUTINE calc_radiation_surface_net(swvis_down, swnir_down, &
2     alb_vis, alb_nir, lw_down, t, rad_net, swvis_net, swnir_net, sw_net, lw_net)
3
4 USE mo_phy_schemes , ONLY: lwnet_from_lwdown
5
6 !$claw model-data
7 REAL(wp), INTENT(in) :: swvis_down, swnir_down, alb_vis, alb_nir, lw_down, t
8 REAL(wp), INTENT(out) :: rad_net
9 REAL(wp), INTENT(out), OPTIONAL :: swvis_net, swnir_net, sw_net, lw_net
10 !$claw end model-data
11
12 REAL(wp) :: zswvis_net, zswnir_net, zsw_net, zlw_net
13
14 ! Compute net SW radiation from downward SW and albedo
15 zswvis_net = swvis_down * (1._wp - alb_vis)
16 zswnir_net = swnir_down * (1._wp - alb_nir)
17 zsw_net = zswvis_net + zswnir_net
18 ! Compute LW net radiation from incoming and the thermal radiation
19 zlw_net = lwnet_from_lwdown(lw_down, t)
20 ! Compute net radiation
21 rad_net = zsw_net + zlw_net
22
23 IF (PRESENT(swvis_net)) swvis_net = zswvis_net
24 IF (PRESENT(swnir_net)) swnir_net = zswnir_net
25 IF (PRESENT(sw_net)) sw_net = zsw_net
26 IF (PRESENT(lw_net)) lw_net = zlw_net
27 END SUBROUTINE calc_radiation_surface_net

```

**Figure 5.** Original source code for an ELEMENTAL subroutine enhanced with CLAW SCA directive inserted by the user

```

1 # CLAW SCA model configuration for JSBACH in ICON
2
3 [model]
4   name = "ICON_JSBACH"
5
6 [[dimensions]]      # Definition of dimensions that can be used in layouts
7   id = "jsb_hori" # Horizontal dimension definition for JSBACH
8   [dimensions.size]
9     lower = 1       # if not specified, 1 by default
10    upper = "nc"   # Number of columns
11
12 [[dimensions]]
13   id = "jsb_soil" # Dimension definition for the number of soil layers
14   [dimensions.size]
15     lower = 1       # Lower bound
16     upper = "nsoil" # Upper boudn for the number of soil layers
17
18 [[layouts]]        # Definition of layouts and default layout for specific target
19   id = "default" # mandatory layout, used if no specific target layout
20           # specified in the sca directive
21   position = [ "jsb_hori", ":" ] # Insert jsb_hori before the existing dimensions
22
23 [[layouts]]
24   id = "nc_nsoil" # Name of the layout
25   # Dimension layout for promotion. jsb_hori and jsb_soil are inserted before
26   # existing dimensions in the given order.
27   position = [ "jsb_hori", "jsb_soil", ":" ]

```

**Figure 6.** CLAW SCA configuration file for ICON model in TOML format

```

1 SUBROUTINE calc_radiation_surface_net ( swvis_down , swnir_down , alb_vis , &
2   alb_nir , lw_down , t , rad_net , swvis_net , swnir_net , sw_net , lw_net )
3
4 USE mo_phy_schemes , ONLY: lwnet_from_lwdown
5
6 REAL ( KIND = wp ) , INTENT(IN) :: swvis_down ( : )
7 REAL ( KIND = wp ) , INTENT(IN) :: swnir_down ( : )
8 REAL ( KIND = wp ) , INTENT(IN) :: alb_vis ( : )
9 REAL ( KIND = wp ) , INTENT(IN) :: alb_nir ( : )
10 REAL ( KIND = wp ) , INTENT(IN) :: lw_down ( : )
11 REAL ( KIND = wp ) , INTENT(IN) :: t ( : )
12 REAL ( KIND = wp ) , INTENT(OUT) :: rad_net ( : )
13 REAL ( KIND = wp ) , INTENT(OUT) :: swvis_net ( : )
14 REAL ( KIND = wp ) , INTENT(OUT) :: swnir_net ( : )
15 REAL ( KIND = wp ) , INTENT(OUT) :: sw_net ( : )
16 REAL ( KIND = wp ) , INTENT(OUT) :: lw_net ( : )
17 REAL ( KIND = wp ) :: zswvis_net
18 REAL ( KIND = wp ) :: zswnir_net
19 REAL ( KIND = wp ) :: zsw_net
20 REAL ( KIND = wp ) :: zlw_net
21 INTEGER :: jsb_hori
22
23 !$acc data @@
24 !$acc present(swvis_down,swnir_down,alb_vis,alb_nir,lw_down,t,rad_net @@
25 !$acc ,swvis_net,swnir_net,sw_net,lw_net)
26 !$acc parallel
27 !$acc loop gang vector
28 DO jsb_hori = 1 , size ( swnir_down , 1 ) , 1
29   zswvis_net = swvis_down ( jsb_hori ) * ( 1._wp - alb_vis ( jsb_hori ) )
30   zswnir_net = swnir_down ( jsb_hori ) * ( 1._wp - alb_nir ( jsb_hori ) )
31   zsw_net = zswvis_net + zswnir_net
32   zlw_net = lwnet_from_lwdown ( lw_down ( jsb_hori ) , t ( jsb_hori ) )
33   rad_net ( jsb_hori ) = zsw_net + zlw_net
34   IF ( present ( swvis_net ) ) THEN
35     swvis_net ( jsb_hori ) = zswvis_net
36 END IF
37   IF ( present ( swnir_net ) ) THEN
38     swnir_net ( jsb_hori ) = zswnir_net
39 END IF
40   IF ( present ( sw_net ) ) THEN
41     sw_net ( jsb_hori ) = zsw_net
42 END IF
43   IF ( present ( lw_net ) ) THEN
44     lw_net ( jsb_hori ) = zlw_net
45 END IF
46 END DO
47 !$acc end parallel
48 !$acc end data
49 END SUBROUTINE calc_radiation_surface_net

```

**Figure 7.** Transformed code with OpenACC directives

target in this case. We rely here on the Fortran compiler to efficiently compile ELEMENTAL on CPU.

When applied to GPU, the following actions occur for non-leaf subroutine or function:

- The signature of the subroutine/function is updated and the ELEMENTAL or PURE specifiers are discarded.
- The flagged fields are promoted according to the specified layout. If no layout is specified, the default layout is assumed. The promotion and layout information come from the configuration file as shown in Fig. 6.
- Iterations over new dimensions specified in the layout are inserted.
- Data analysis is performed and temporary fields or scalars that need promotion are promoted.

Leaf ELEMENTAL subroutines and functions have simpler transformations applied to them. The code is processed as follows:

- As for other ELEMENTAL subroutines/functions, the signature is updated and the ELEMENTAL or PURE specifiers are discarded.
- Compiler directives are inserted to set the subroutine or function as an OpenACC routine or OpenMP target code.

Once the code is annotated with CLAW SCA directives, the CLAW Compiler is called the same way for file with SCA or SCA with ELEMENTALs. Listing 4 can be executed in the same way.

### 3.1. Expansion Directive

As mentioned in Section 1, vector notation is used widely in tasks. To automatize the port of these blocks of Fortran code to OpenACC and OpenMP, the CLAW expand directive is used.

```

1 !$claw expand parallel
2   rad_srf_net(:) = (1._wp - fract_lice(:)) * rad_net_lwtr(:) + fract_lice(:) *
3     rad_net_lice(:)
4   sw_srf_net(:) = (1._wp - fract_lice(:)) * sw_net_lwtr(:) + fract_lice(:) *
5     sw_net_lice(:)
6   lw_srf_net(:) = (1._wp - fract_lice(:)) * lw_net_lwtr(:) + fract_lice(:) *
7     lw_net_lice(:)
8 !$claw end expand

```

**Figure 8.** Block of vector notation with CLAW expand directive

```

1 !$acc parallel loop gang vector DO claw_induction_0 = 1 , size ( rad_srf_net , 1
2   )
3   rad_srf_net ( claw_induction_0 ) = ( 1._wp - fract_lice ( claw_induction_0 &
4     ) ) * rad_net_lwtr ( claw_induction_0 ) + fract_lice ( claw_induction_0 ) &
5     * rad_net_lice ( claw_induction_0 )
6   sw_srf_net ( claw_induction_0 ) = ( 1._wp - fract_lice ( claw_induction_0 &
7     ) ) * sw_net_lwtr ( claw_induction_0 ) + fract_lice ( claw_induction_0 ) &
8     * sw_net_lice ( claw_induction_0 )
9   lw_srf_net ( claw_induction_0 ) = ( 1._wp - fract_lice ( claw_induction_0 &
10    ) ) * lw_net_lwtr ( claw_induction_0 ) + fract_lice ( claw_induction_0 ) &
     * lw_net_lice ( claw_induction_0 ) END DO

```

**Figure 9.** Expanded vector notation

Figure 8 presents a block of vector notation operations surrounded by a CLAW expand block. The expand block works only for specific target as the CLAW SCA for ELEMENTAL. Before the transformation is applied to the block, an analysis is performed to make sure all the array dimensions are compatible to be transformed within the same loop structure. If the criteria are met, iteration are inserted on appropriate range and parallelization for the given directives. Figure 9 is the same code after transformation applied to it.

## 4. Performance Comparison

In this section we present the performance results that have been achieved by applying the extended version of the CLAW Compiler to the JSBACH land surface scheme. The computational domain size (number of horizontal grid points  $\times$  number of soil layers) used for the performance measurement is  $20480 \times 5$ . All performance measurements in this section are obtained using  $-03$  optimization flag or equivalent for each compiler. The code is transformed with the CLAW Compiler 2.0 and compiled with the PGI 18.10 Fortran compiler for the baseline

CPU as well as the GPU OpenACC results. The CPU reference is obtained using multi-core OpenMP parallelism available in the original version of ICON. The GPU OpenMP target results were obtained with Cray Compiler CCE 8.7.4 and by running standalone version of the kernels since the full ICON model is not ready to have a code mixing OpenMP for multi-core and for accelerator.

Figure 10 shows the speedup achieved from the CLAW SCA transformed version with OpenACC directives and the CLAW SCA transformed version with OpenMP directives over the CPU reference. The original version of the code is exploiting the 12 cores available on the Intel Xeon E5-2690 v3 Haswell CPU of Piz Daint at CSCS parallelized with multi-core OpenMP. The CLAW OpenACC and OpenMP versions are executed on one NVIDIA P100 GPU. The theoretical floating point peak performance of the Intel Haswell is 500 GFLOPS, while the NVIDIA P100 is 5.3 TFLOPS. In term of memory bandwidth, the Haswell has a theoretical peak at 68 GB/s while the P100 can reach up to 732 GB/s. For both compute and memory bandwidth limited problem one can expect a maximum speedup of 11x.

As shown, OpenACC and OpenMP results are very similar and expected. Kernels issued from the ELEMENTAL transformation are pretty simple to be handled by a compiler and we did not expect PGI and Cray to fundamentally generate different code for them. Depending on the size of the kernel and the tile it has to process, we are able to achieve speedup between 1.7x up to 8.6x for kernel like the albedo: calc\_sky\_view\_fractions . Tiles can be viewed as a collection of grid points from a specific type of land (e.g., lake or glacier) or as a mask for this specific type of land. Therefore tiles can have more or less work to be performed due to a different set of grid points. As GPUs need enough work to exploit massive parallelism, a reduced set of grid points is one of the reason the speedup can vary this much. Another factor is the size of the kernel. Some ELEMENTAL functions/subroutines are very small and the kernels generated from them are also small. Kernel's launch and synchronization is then a non-negligible part of the overall kernel execution time. These overhead can be hidden in the execution of the full model with asynchronous mechanism.

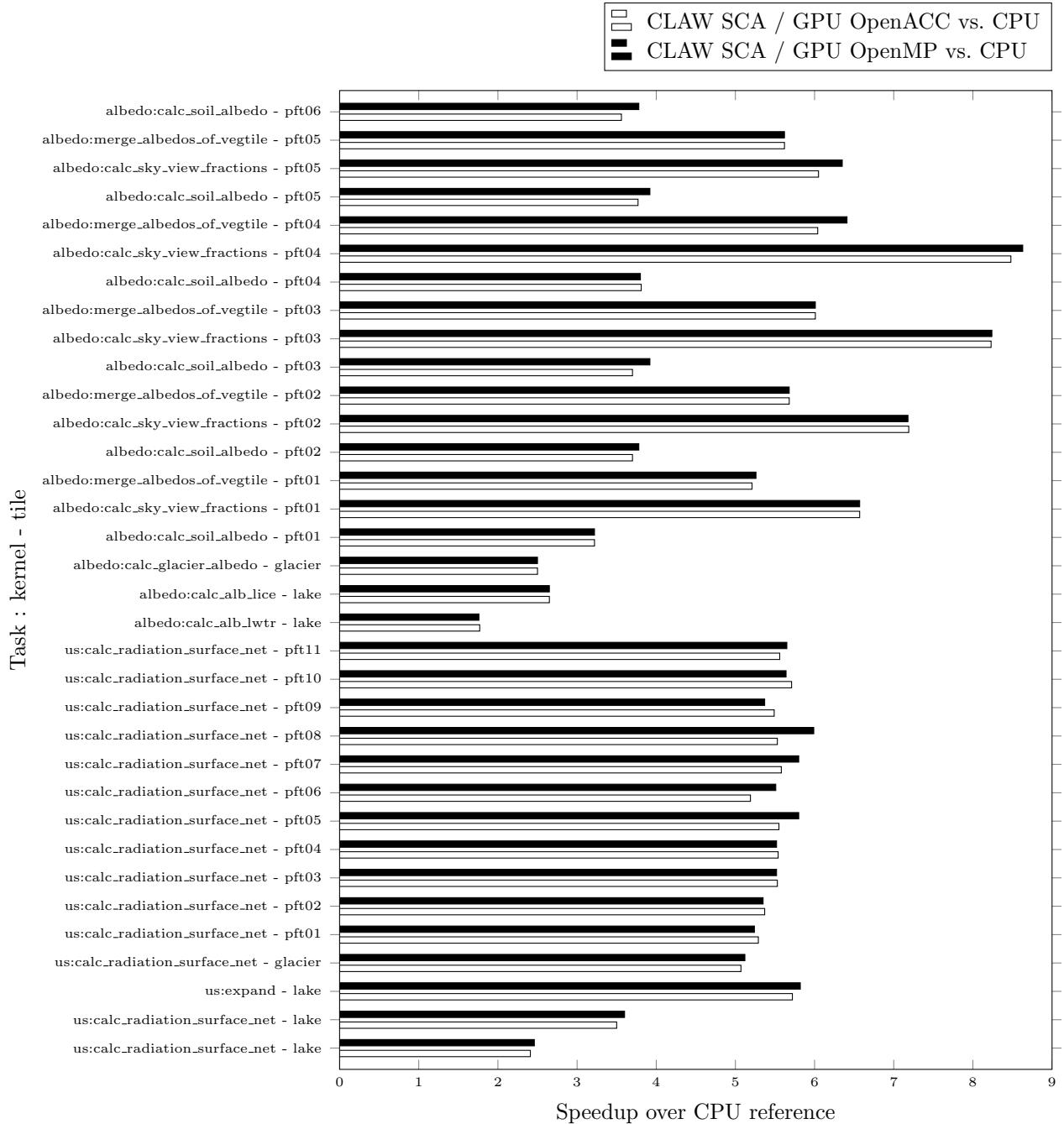
## 5. Code Metrics

This section briefly compares the original code with its CLAW SCA version before and after applying transformations. Unlike the original SCA the SCA for ELEMENTAL imposes less changes to the code. In order to comply with the specification, the user only annotates the ELEMENTAL subroutines and functions to be executed on GPU, and relies on the compiler from then on. There is no need to apply other code change in the ELEMENTALs.

The full JSBACH model is annotated with 40 CLAW SCA directives and about 130 CLAW expansion directives to convert vector notation to parallelizable loops. In the transformed code there are almost 2000 lines of OpenACC directives or OpenMP depending the user's choice. This is the amount of compiler directives a user would have had to write by hand to achieve the same goal. A hand-written version would also have to change the code structure significantly, for example by promoting fields and adding loops.

## Conclusion

In this paper we propose an extension of the CLAW SCA DSL and its compiler to take advantage of the ELEMENTAL construct in Fortran code and automatize the port to heteroge-



**Figure 10.** Performance comparison (socket to socket) between CLAW OpenACC, CLAW OpenMP and the CPU reference of kernels from two JSBACH tasks on Intel Haswell E5-2690v3 and NVIDIA P100. Domain size (number of horizontal grid points × number of soil layers) = 20480×5

nous architecture. The ELEMENTAL construct as exploited in JSBACH provides the necessary abstraction to implement automatic code transformation and target GPU architectures without writing lots of compiler directives by hand. Promotion, loop parallelization and compiler directive generation are handled by the CLAW Compiler automatically.

For this paper, the CLAW SCA extension was applied to a wide portion of the JSBACH land surface scheme, one of the physical parametrizations of the ICON global climate model. From a single simple source code multiple programming paradigm can be targeted. Performance

results show up to 8.6x speedup against a 12-core parallel CPU version for a specific kernel. All kernels are at least 1.7x faster than the CPU version. The overall performance of the JSBACH model running on GPU is typically between 5x to 6x speedup depending on its configuration.

In the current implementation of the extension, a single parallel subroutine or function is generated from its ELEMENTAL counterpart. This is fine for the JSBACH use case as a single ELEMENTAL is always called with the same kind of arguments. Future improvement to generate several versions of a subroutine or a function if the type of argument used to call them is different.

As it is possible to generate any source code, we can imagine to take advantage of new compiler development such as exploiting the **DO CONCURRENT** construct from Fortran 2008 to target accelerators as it is investigated in latest version of PGI. Instead of generating compiler directives the CLAW Compiler could exploit this new Fortran feature.

## Acknowledgements

This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID c15. We would like to show our gratitude to the OMNI Compiler Team at RIKEN CCS for their effort and support to provide a very useful open-source tool-chain. We would also like to thank Reiner Schnur, a scientific programmer at the Max-Plank-Institut für Meteorologie for his insights in the JSBACH code base and review of our changes. This work was partly funded by ETH Zurich and the PASC Initiative under the ENIAC project.

*This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.*

## References

1. Clement, V., Ferrachat, S., Fuhrer, O., et al.: The CLAW DSL: Abstractions for performance portable weather and climate models. In: Proceedings of the Platform for Advanced Scientific Computing Conference. pp. 2:1–2:10. PASC ’18, ACM, New York, NY, USA (2018), DOI: 10.1145/3218176.3218226
2. Crueger, T., Giorgetta, M.A., Brokopf, R., et al.: ICON-A, the atmosphere component of the ICON Earth system model: II. model evaluation. Journal of Advances in Modeling Earth Systems 10(7), 1638–1662 (2018), DOI: 10.1029/2017MS001233
3. Fuhrer, O., Osuna, C., Lapillonne, X., et al.: Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. Supercomputing Frontiers and Innovations 1(1), 45–62 (2014), DOI: 10.14529/jsfi140103
4. Giorgetta, M.A., Brokopf, R., Crueger, T., et al.: ICON-A, the atmosphere component of the ICON Earth system model: I. model description. Journal of Advances in Modeling Earth Systems 10(7), 1613–1637 (2018), DOI: 10.1029/2017MS001242
5. Gysi, T., Osuna, C., Fuhrer, O., et al.: Stella: A domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for

High Performance Computing, Networking, Storage and Analysis. pp. 41:1–41:12. SC '15, ACM, New York, NY, USA (2015), DOI: 10.1145/2807591.2807627

6. Lapillonne, X., Fuhrer, O.: Using compiler directives to port large scientific applications to GPUs: An example from atmospheric science. *Parallel Processing Letters* 24(1) (2014), DOI: 10.1142/S0129626414500030
7. Mauritsen, T., Bader, J., Becker, T., et al.: Developments in the MPI-M Earth System Model version 1.2 (MPI-ESM1.2) and Its Response to Increasing CO<sub>2</sub>. *Journal of Advances in Modeling Earth Systems* 11(4), 998–1038 (2019), DOI: 10.1029/2018MS001400
8. Muller, M., Aoki, T.: Hybrid Fortran: High productivity GPU porting framework applied to Japanese weather prediction model. *CoRR* abs/1710.08616 (2017), <http://arxiv.org/abs/1710.08616>
9. Omni CompilerProject: Omni Compiler Project - An Infrastructure for Source-to-Source Transformation. <http://omni-compiler.org> (2013-2019), accessed: 2019-09-02
10. OpenACC Standard: The OpenACC application programming interface - version 2.7. <https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf> (2018), accessed: 2019-09-02
11. OpenMP Architecture Review Board: OpenMP application programming interface - version 5.0. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf> (2018), accessed: 2019-09-03
12. XcalableMP Specification Working Group: XcodeML/Fortran Specification. <https://omni-compiler.org/download/xcodeml/stable/XcodeML-F-1.0.pdf> (2017), accessed: 2019-09-02

# Optimizing Deep Learning RNN Topologies on Intel Architecture

*Kunal Banerjee<sup>1</sup>, Evangelos Georganas<sup>2</sup>, Dhiraj D. Kalamkar<sup>1</sup>,  
Barukh Ziv<sup>3</sup>, Eden Segal<sup>3</sup>, Cristina Anderson<sup>4</sup>, Alexander Heinecke<sup>2</sup>*

© The Authors 2019. This paper is published with open access at SuperFri.org

Recurrent neural network (RNN) models have been found to be well suited for processing temporal data. In this work, we present an optimized implementation of vanilla RNN cell and its two popular variants: LSTM and GRU for Intel Xeon architecture. Typical implementations of these RNN cells employ one or two large matrix multiplication (GEMM) calls and then apply the element-wise operations (sigmoid/tanh) onto the GEMM results. While this approach is easy to implement by exploiting vendor-optimized GEMM library calls, the data reuse relies on how GEMMs are parallelized and is sub-optimal for GEMM sizes stemming from small minibatch. Also, the element-wise operations are exposed as a bandwidth-bound kernel after the GEMM which is typically a compute-bound kernel. To address this discrepancy, we implemented a parallel blocked matrix GEMM in order to (a) achieve load balance, (b) maximize weight matrix reuse, (c) fuse the element-wise operations after partial GEMM blocks are computed and while they are hot in cache. Additionally, we bring the time step loop in our cell to further increase the weight reuse and amortize the overhead to transform the weights into blocked layout. The results show that our implementation is generally faster than Intel MKL-DNN library implementations, e.g. for RNN, forward pass is up to  $\sim 3\times$  faster whereas the backward/weight update pass is up to  $\sim 5\times$  faster. Furthermore, we investigate high-performance implementations of sigmoid and tanh activation functions that achieve various levels of accuracy. These implementations rely on minimax polynomial approximations, rational polynomials, Taylor expansions and exponential approximation techniques. Our vectorized implementations can be flexibly integrated into deep learning computations with different accuracy requirements without compromising performance; in fact, these are able to outperform vectorized and reduced accuracy vendor-optimized (Intel SVML) libraries by  $1.6\text{--}2.6\times$  while speed up over GNU libm is close to two orders of magnitude. All our experiments are conducted on Intel's latest CascadeLake architecture.

*Keywords:* LSTM, Intel Xeon, GEMM, compute-bound kernel, bandwidth-bound kernel.

## Introduction

RNN models, unlike typical feed-forward artificial neural network models, allow connections between nodes to form a directed graph along a temporal sequence and hence, by design, are well equipped to learn from temporal data. These models have found applications in language translation [24], text generation [23], handwriting recognition [15] and image captioning [8] among many others. A particular variant of RNN called long short-term memory (LSTM) provides improvements over traditional RNN by handling exploding and vanishing gradient problems encountered during RNN training [16]. Another variant of RNN called gated recurrent unit (GRU) has been proposed which has fewer parameters than LSTM [10]. However, the choice between LSTM and GRU is not always clear and may depend on the dataset and/or the task at hand [10]. Therefore, it is important to have efficient implementations of the aforementioned RNN models in order to expedite training and inference of the various RNN based applications, especially, if these applications undergo continuous learning and/or deployed in scenarios where these are expected to perform real-time predictions.

<sup>1</sup>Intel Corporation, Bangalore, India

<sup>2</sup>Intel Corporation, Santa Clara, USA

<sup>3</sup>Intel Corporation, Haifa, Israel

<sup>4</sup>Intel Corporation, Oregon, USA

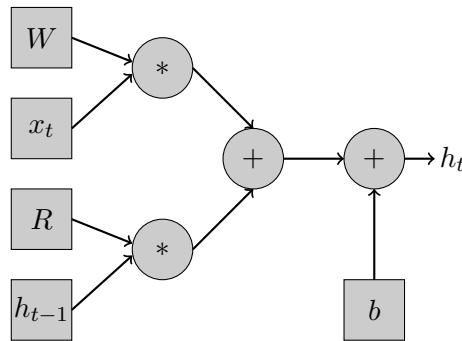
Let us now dive into the RNN cell. It is pertinent to note that non-linear activation functions, such as tanh and sigmoid (we use the term *sigmoidal* in this paper to refer to either of these two activation functions) help with the generalization of the models and the differentiation between the outputs. However, these functions are computationally expensive even in single precision since their definitions require the calculation of the exponential function. To make things even worse, emerging deep-learning hardware focuses mostly on the acceleration of the GEMM-flavored computational kernels and consequently the fraction of the time spent in such non-linear activation functions becomes even larger. Nevertheless, the current trend in deep learning is to use lower precision (e.g. float16, bfloat16 [1], int16, int8) for activations in both inference and training [12, 13] and as a result high-performance, reduced precision or lower accuracy activation functions are a viable option.

In this paper, we present implementation of a vanilla RNN cell (with support for different non-linearities) and the two popular variants, namely LSTM and GRU, which are specifically tuned for Intel Xeon architecture. In the process, we showcase benefits of our approach over Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN) [2] implementation of these RNN variants, which is an open source performance library from Intel, intended for acceleration of deep learning frameworks on Intel architecture. Moreover, we focus on high-performance and reduced precision implementations of sigmoidal functions that are a natural fit for RNNs and deep learning, in general. In particular, we investigate approximation techniques based on: (a) Padé rational polynomials [7], (b) piecewise minimax polynomials [20] and (c) approximations of the exponential function via Taylor expansions [11]. Our implementations are vectorized with AVX512 instructions targeting modern Intel CPUs. We additionally explore the trade-off between accuracy and speed of these approximations. All our code is publicly available at [3] as part of the LIBXSMM library.

Rest of this paper is organized as follows. Section 1 provides an overview of the RNN cell and its variants along with the details of the steps taken to optimize these on Intel Xeon architecture. Section 2 describes the various approximation algorithms for sigmoidal functions along with their implementations. Section 3 covers the experimental results; specifically, we characterize the performance of our cell implementations vis-a-vis those of MKL-DNN; we further showcase a comparative analysis of the various approximations of sigmoidal functions on Intel Xeon architecture.

## 1. Implementation of RNN Cell and its Variants

### 1.1. (Vanilla) RNN Cell



**Figure 1.** A diagram of an RNN cell

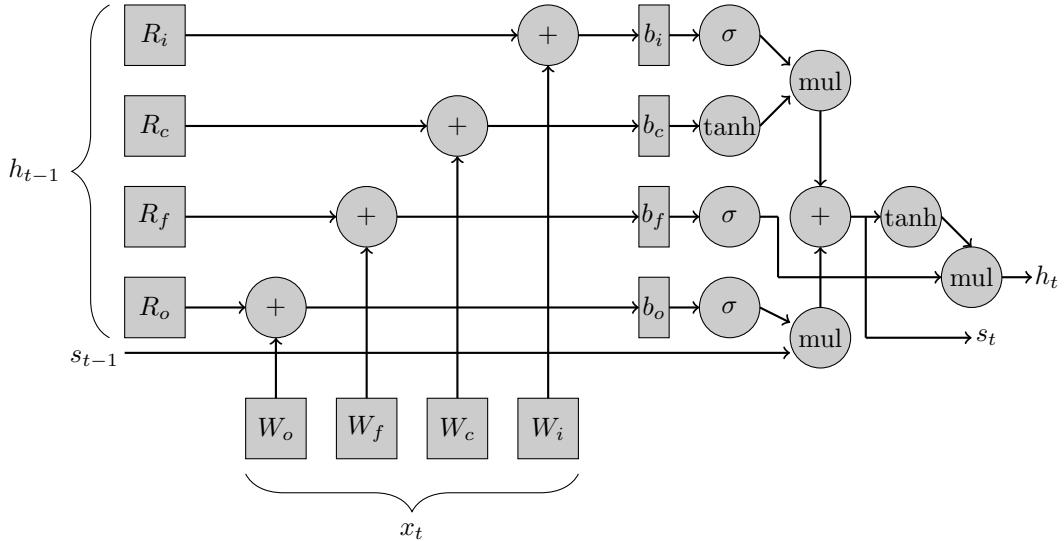
RNN models have found staggering success in learning from temporal data as documented in [18]. A diagram of an RNN cell is shown in Fig. 1. The equation representing one forward step of an RNN cell from timestep  $t - 1$  to  $t$  is given below:

$$h_t = \Lambda(W * x_t + R * h_{t-1} + b),$$

where  $h$  is the hidden state of the RNN,  $x$  is the input from the previous layer,  $W$  is the weight matrix for the input,  $R$  is the weight matrix for the recurrent connections,  $b$  is the bias and  $\Lambda$  is a non-linear function which can be ReLU or tanh or sigmoid ( $\sigma$ ) (while the former two nonlinearities are supported by cuDNN [9], the last one has been used by Baidu [14] – we support all three variants in our implementation; note that the backpropagation equations depend on the choice of  $\Lambda$ ).

Typically, RNN implementations involve two GEMMs:  $W * x$  and  $R * h$ , or one GEMM: concatenated  $WR$  with concatenated  $xh$ . The GEMM operation is followed by application of element-wise operation  $\Lambda$  on the GEMM result. While this approach is easy to implement by exploiting vendor-optimized GEMM library calls, the data reuse relies on how GEMMs are parallelized and is sub-optimal for GEMM sizes stemming from small minibatch. Also, the element-wise operations are exposed as a bandwidth-bound kernel after the GEMM which is typically a compute-bound kernel. To address this inconsistency, our optimization of the LSTM cell is based on a “data-flow” approach. We implemented a parallel blocked matrix GEMM in order to (a) achieve load balance, (b) maximize weight matrix reuse and (c) fuse the element-wise operations after partial GEMM blocks are computed and while they are hot in cache. Internally, we use a blocked matrix layout for weights and traditional activation format so that we exploit a better locality and avoid conflict misses. Additionally, we bring the time step LSTM loop in our cell to further increase the weight reuse and amortize the overhead to transform the weights into the blocked layout. Algorithm 1 captures the method described above succinctly.

## 1.2. LSTM Cell



**Figure 2.** A diagram of an LSTM cell

A diagram of an LSTM cell is given in Fig. 2. The equations to compute the output of the LSTM cell at time step  $t$  are as follows:

**Algorithm 1** Forward propagation pass of RNN cell

---

**Inputs:**  $N$ : batch size,  $C$ : input channel size,  $K$ : output channel size,  $T$ : time steps,

Weight tensors  $W$ ,  $R$ , bias  $b$ , input sequences  $x$ ,  $h$ , blocking factors  $b_N$ ,  $b_C$ ,  $b_K$ 
**Output:** Output sequence  $h$ 

```

1: Convert  $W$ ,  $R$ ,  $x_t$ ,  $h_{t-1}$  into blocked format. //cf. Section 3.1
2: Based on thread_id  $i$ , calculate corresponding output block of  $h_t$ : let its corner indices be
    $N_i^{start}$ ,  $N_i^{end}$ ,  $K_i^{start}$ ,  $K_i^{end}$ ; let  $C_i^{start}$  and  $C_i^{end}$ , and  $\mathcal{K}_i^{start}$  and  $\mathcal{K}_i^{end}$  be the corresponding
   corner indices along input channels and output channels, correspondingly, in  $x_t$ ,  $W$  and  $R$ 
   which are required to compute  $h_t$ .
3: for  $t = 0 \dots T - 1$  {
4:   for  $k = K_i^{start} \dots K_i^{end}$  {
5:     for  $n = N_i^{start} \dots N_i^{end}$  {
6:       for  $c = C_i^{start} \dots C_i^{end}$  {
7:          $\mathcal{A} \leftarrow \text{blocked\_GEMM}(W[k][c][b_C][b_K], x[t][n][c][b_N][b_C]).$ 
8:       }
9:       for  $c = \mathcal{K}_i^{start} \dots \mathcal{K}_i^{end}$  {
10:         $\mathcal{B} \leftarrow \text{blocked\_GEMM}(R[k][c][b_K][b_K], h[t][n][c][b_N][b_C]).$ 
11:      }
12:       $h[t + 1][n][c][b_N][b_C] \leftarrow \Lambda(\mathcal{A} + \mathcal{B} + b[k]).$ 
13:    }
  }
}

```

---

$$\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i), \\
c_t &= \tanh(W_c * x_t + R_c * h_{t-1} + b_c), \\
f_t &= \sigma(W_f * x_t + R_f * h_{t-1} + b_f), \\
o_t &= \sigma(W_o * x_t + R_o * h_{t-1} + b_o), \\
s_t &= f_t \circ s_{t-1} + i_t \circ c_t, \\
h_t &= o_t \circ \tanh(s_t).
\end{aligned}$$

Typical implementations of LSTM involve two large GEMMs:  $\mathbf{W} * \mathbf{x}$  and  $\mathbf{R} * \mathbf{h}$ , where  $\mathbf{W}$  and  $\mathbf{R}$  are obtained by concatenating  $W_i, W_c, W_f, W_o$  and  $R_i, R_c, R_f, R_o$  respectively, or one larger GEMM: concatenated  $\mathbf{WR}$  with concatenated  $\mathbf{xh}$ , for example, original implementation of LSTM cell in TensorFlow has adopted this one larger GEMM approach. This GEMM step is followed by application of element-wise operations (sigmoid/tanh) on the GEMM results. Our implementation of LSTM cell follows the same principle as in Algorithm 1 with the exception that steps 6–11 are repeated four times (one each for  $i$ ,  $c$ ,  $f$  and  $o$ ), and instead of single step 12, there are multiple element-wise operations which are applied to compute  $s$  and  $h$ .

### 1.3. GRU Cell

It is important to note that we have found subtle differences in the equations for GRU across different implementations, e.g., those between cuDNN [9] and TensorFlow [5]. We have adopted the formulation provided in TensorFlow (to aid in future integration) as stated below:

$$\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i), \\
c_t &= \sigma(W_c * x_t + R_c * h_{t-1} + b_c), \\
o_t &= h_{t-1} \circ i_t, \\
f_t &= \tanh(W_f * x_t + R_f * o_t + b_f),
\end{aligned}$$

$$h_t = (1 - c_t) \circ f_t + c_t \circ h_{t-1}.$$

We have followed the same design policies as mentioned for LSTM and hence we refrain from further elaboration for brevity.

## 2. Implementations for Approximate Sigmoidal Functions

For neural networks, activation functions comprise an important feature that essentially determines whether neurons in the network should be activated or not. The most widely used activation functions are nonlinear like tanh and sigmoid because such functions help with the generalization of the models and the differentiation between the outputs. However, these functions are computationally expensive because their definitions require the calculation of the exponential function. Therefore, it is worth investigating algorithms which can approximate these non-linear activation functions without sacrificing accuracy considerably. In this work, we focus on algorithms which can be easily implemented in software with Intel's existing 512bit SIMD instruction set, and consequently, do not require any specialized hardware, such as the ones reported in [19, 22].

For the remainder of this paper, we will be considering only the tanh function given that the standard logistic sigmoid is a rescaled tanh:

$$\text{sigmoid}(x) = (\tanh(x/2) + 1)/2.$$

### 2.1. Rational Padé Approximations

The tanh function has two asymptotes, therefore approximating it with a mere Taylor expansion of low degree would result in poor approximation. Instead we consider rational approximations of tanh and more specifically the Padé rational polynomials. The rational Padé approximation  $\text{Padé}_{[p/q]}f(x)$  of a function  $f$  is a ratio of two polynomials with degrees  $p$  and  $q$ :

$$\text{Padé}_{[p/q]}f(x) = \frac{\sum_{i=0}^p a_i x^i}{\sum_{i=0}^q b_i x^i}.$$

which agrees with  $f$  to the highest possible order, in essence:

$$\begin{aligned} f(0) &= \text{Padé}_{[p/q]}f(0), \\ f'(0) &= \text{Padé}'_{[p/q]}f(0), \\ &\vdots \\ f^{(p+q)}(0) &= \text{Padé}_{[p/q]}^{(p+q)}f(0). \end{aligned}$$

To calculate the coefficients  $a_i$  and  $b_i$  one can consider the first  $p + q$  derivatives of  $f$  at zero and finally solve the corresponding system of equations. By construction, the error of  $\text{Padé}_{[p/q]}f(x)$  agrees with the truncation error of the Taylor series about 0 truncated at the  $(p + q)^{\text{th}}$  term.

The implementation of the  $\text{Padé}_{[3/2]}(x)$  for tanh with AVX512 instructions is shown in Fig. 3 (for simplicity we do not include the initialization of the constant vectors). This implementation uses Fused Multiply and Accumulate (FMA) instructions to evaluate the polynomials in the numerator and the denominator of the rational approximation via the Horner's rule. For performance reasons, instead of dividing the numerator by the denominator, we use the approximate

reciprocal intrinsic (line 5 in Fig. 3) and a multiplication. Also, in order to ensure that the function is bound within the asymptotes at  $y = \pm 1$ , we clip it for input values beyond two limits  $lo$  and  $hi$  by leveraging two compare and two blend instructions (lines 7–10). In Section 3.2, we also include the evaluation of the  $\text{Padé}_{[7/8]}(x)$  approximation which performs more FMAs for the respective polynomial evaluations but also achieves higher accuracy.

```

__m512 x2      = _mm512_mul_ps(x, x);           //line 1
__m512 t1_nom  = _mm512_fmadd_ps(x2, c1, one);   //line 2
__m512 nom     = _mm512_mul_ps(t1_nom, x);         //line 3
__m512 denom   = _mm512_fmadd_ps(x2, c2, one);       //line 4
__m512 denom_rcp = _mm512_rcp14_ps(denom);        //line 5
__m512 result   = _mm512_mul_ps(nom,denom_rcp);    //line 6
__mmask16 maskHi = _mm512_cmp_ps_mask(x, hi, _CMP_GT_OQ); //line 7
__mmask16 maskLo = _mm512_cmp_ps_mask(x, lo, _CMP_LT_OQ); //line 8
result = _mm512_mask_blend_ps(maskHi, result, one); //line 9
result = _mm512_mask_blend_ps(maskLo, result, neg_one); //line 10

```

**Figure 3.** Rational Padé 3/2 approximation

## 2.2. Piecewise Minimax Polynomials Approximations

Another approach for approximating transcendental functions is to use piecewise polynomial approximations. Various polynomials may be chosen for approximation; in this section we leveraged minimax polynomials [20]. Therefore, we divide the input range of the function  $\tanh(x)$  into intervals and for each interval  $[a, b]$  we find a polynomial  $p$  of degree at most  $n$  to minimize:

$$\max_{a \leq x \leq b} |\tanh(x) - p(x)|.$$

In our approximations, we utilize truncated Chebyshev series [21] that closely approximate the minimax polynomials.

Figure 4 shows the implementation of the  $\tanh$  function approximation using minimax polynomials of second degree. First, the function’s input range is divided into 16 intervals using the argument’s exponent and the Most Significant Bit (MSB) and an index register `idx` is generated (lines 2–5). Then, the code uses 3 lookup tables (`tanh_c0_reg`, `tanh_c1_reg` and `tanh_c2_reg`) consisting of 16 entries each to simultaneously load 16 triples of coefficients of the approximating polynomial. In order to load the coefficients in three registers, we utilize the `_mm512_permutexvar_ps` intrinsic which shuffles single-precision values in a zmm register using the corresponding index `idx` (lines 6–8). This in-register look-up table is much faster ( $\sim 4\times$ ) than using AVX512’s gather instructions from memory. The polynomial evaluation is materialized with FMAs and the Horner’s rule. Only the positive range of inputs is represented; meanwhile, for negative input values, we exploit the property  $\tanh(-x) = -\tanh(x)$  resulting from the point symmetry (line 11).

There are two ways to reduce the approximation error:

*Divide the input range to more intervals:* Increasing the number of intervals from 16 to 32 will require 3 additional zmm registers to hold the coefficients. The number of instructions will not change but the loading of the coefficients will have a greater latency because we have to use the `_mm512_permutex2var_ps` instruction which shuffles single-precision elements in two zmm

registers using the corresponding selector/index in `idx`.

*Use polynomials of higher degree:* Every additional degree will need an additional zmm register to store the coefficients, one additional instruction to load the coefficients and one additional FMA to evaluate the polynomial. In Section 3.2, we also evaluate the tanh approximation using minimax polynomials of third degree.

```

__m512 signs = _mm512_and_ps(x, ps_sign_mask); //line 1
__m512 abs_x = _mm512_and_ps(x, ps_sign_filter); //line 2
__m512i idx = _mm512_srli_epi32(_mm512_castps_si512(abs_x),22); //line 3
idx = _mm512_max_epi32(idx, lut_low); //line 4
idx = _mm512_min_epi32(idx, lut_high); //line 5
__m512 c0 = _mm512_permutexvar_ps(idx, tanh_c0_reg); //line 6
__m512 c1 = _mm512_permutexvar_ps(idx, tanh_c1_reg); //line 7
__m512 c2 = _mm512_permutexvar_ps(idx, tanh_c2_reg); //line 8
__m512 result = _mm512_fmadd_ps(abs_x, c2, c1); //line 9
result = _mm512_fmadd_ps(abs_x, result, c0); //line 10
result = _mm512_xor_ps(result, signs); //line 11

```

**Figure 4.** Approx. with 2<sup>nd</sup> degree minimax polynomials

### 2.3. Tanh via exp() Approximation with Taylor Series

In this approximation method (see Fig. 5), we use the definition of tanh:

$$\tanh(x) = 1 - \frac{2}{1 + e^{2x}}$$

and we approximate the calculation of  $e^{2x}$ .

In order to approximate the function  $e^x$ , we exploit the property  $e^x = 2^{x \log_2 e} = 2^{n+y} = 2^n \cdot 2^y$  with  $n = \text{round}(x \log_2 e)$  and  $y = x \log_2 e - n$ . With these properties in mind, all we have to do is to compute the term  $2^n$  with  $n$  being an integer, and the term  $2^y$  with  $|y| \in [0, 1)$ . For the term  $2^y$  we use a second-degree Taylor polynomial (lines 3-4). Once  $2^y$  is calculated, we leverage the instruction `_mm512_scalef_ps(A,B)` which returns a zmm holding  $a_i \cdot 2^{\text{floor}(b_i)}$  for each  $a_i \in A$  and  $b_i \in B$ . This scale instruction (line 5) concludes the approximation of the exponential part in the denominator. The approximation error can be further reduced by using a larger-degree Taylor expansion for the  $2^y$  term; in Section 3.2, we also evaluate a third-degree Taylor polynomial.

```

__m512 _x = _mm512_fmadd_ps(x, twice_log2_e, half); //line 1
__m512 y = _mm512_reduce_ps(_x, 1); //line 2
__m512 t1 = _mm512_fmadd_ps(y, c2, c1); //line 3
__m512 two_to_y = _mm512_fmadd_ps(y, t1, c0); //line 4
__m512 exp = _mm512_scalef_ps(two_to_y, _x); //line 5
__m512 den_rcp = _mm512_rcp14_ps(_mm512_add_ps(exp, one)); //line 6
__m512 result = _mm512_fmadd_ps(den_rcp, minus_two, one); //line 7

```

**Figure 5.** Tanh via exp() approximation

### 3. Experimental Results

#### 3.1. Evaluation of RNN Cell and its Variants

Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN) is an open source performance library from Intel intended for acceleration of deep learning frameworks on Intel architecture. Hence, we choose to compare our LSTM cell with that of MKL-DNN (version 1.0.2). We also compare it with BLAS where we perform  $\mathbf{W} * \mathbf{x}$  and  $\mathbf{R} * \mathbf{h}$  followed by element-wise operations for the forward pass; note that we cannot concatenate weights and perform a single large BLAS call (with enhanced performance) in the backward/weight update pass and hence we omit comparison with BLAS during this pass. Note that all the numbers reported in this subsection are measured on Intel® Xeon® Platinum 8280 processor codenamed **CascadeLake (CLX)** with 28 cores at 2.4 GHz AVX turbo frequency.

##### 3.1.1. RNN cell efficiency

As mentioned earlier, we have adopted a “dataflow”-based approach for optimizations. We use blocked layout to better exploit locality and avoid conflict misses. Given  $N$  = minibatch size,  $C$  = input channels and  $K$  = output channels and  $T$  = total time steps, internally, we transform the inputs in blocked format as mentioned below:

- input activations:  $[T][N][C] \rightarrow [T][N/B_N][C/B_C][B_N][B_C]$ ;
- hidden activations:  $[T][N][K] \rightarrow [T][N/B_N][K/B_K][B_N][B_K]$ ;
- weights:  $[C][K] \rightarrow [K/B_K][C/B_C][B_C][B_K]$ ;
- recurrent weights:  $[K][K] \rightarrow [K/B_K][K/B_K][B_K][B_K]$ ,

where  $B_N$ ,  $B_C$  and  $B_K$  are blocking factors for  $N$ ,  $C$  and  $K$ , respectively. We perform computation with fused-time steps which amortizes the cost of blocking.

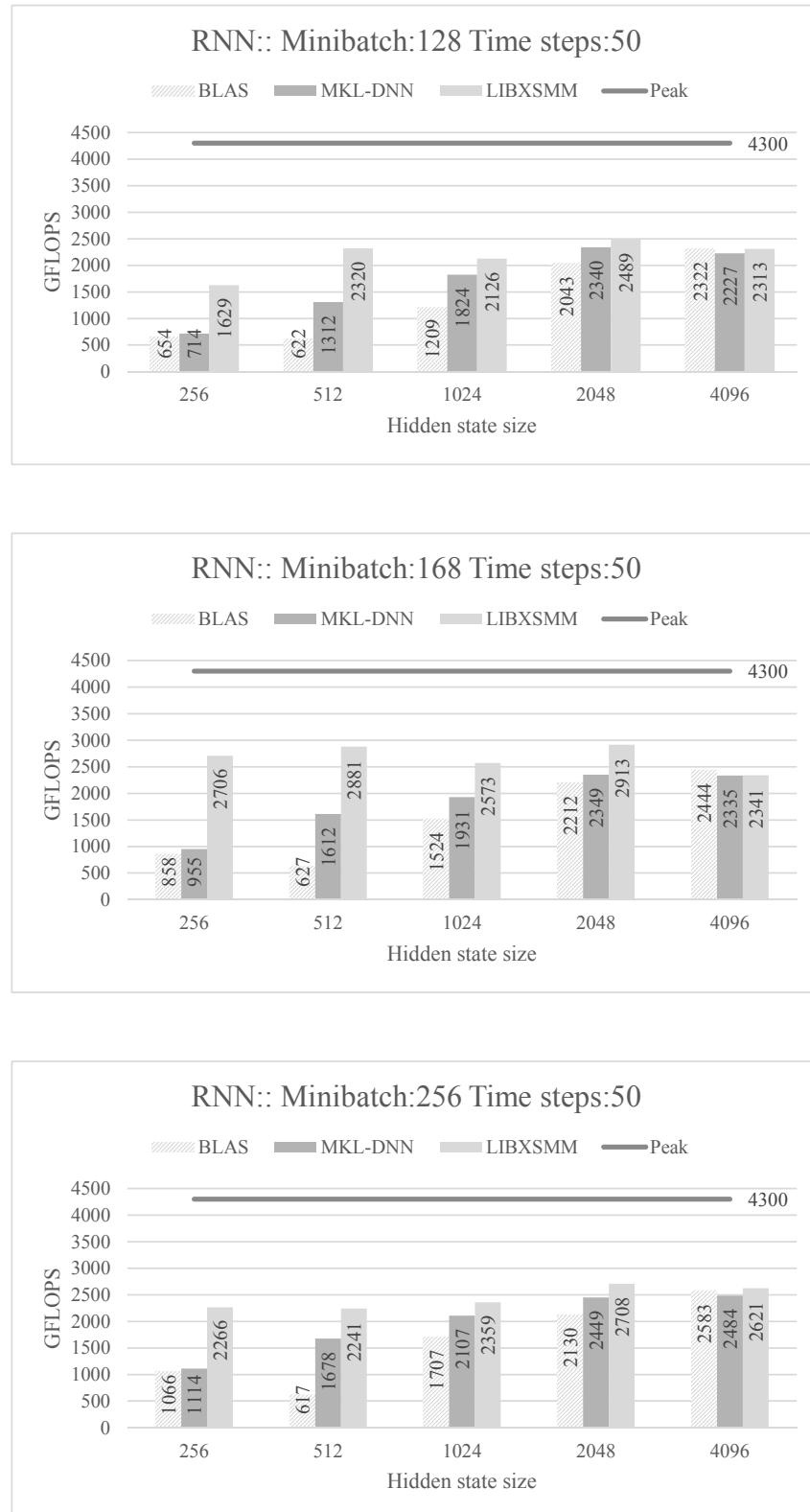
The heart of our blocked matrix GEMM consists of a JIT-ed small batch-reduce GEMM kernel which we implemented in LIBXSMM. The batch-reduce GEMM materializes the operation

$$C = \sum_{i=1}^n A_i * B_i$$

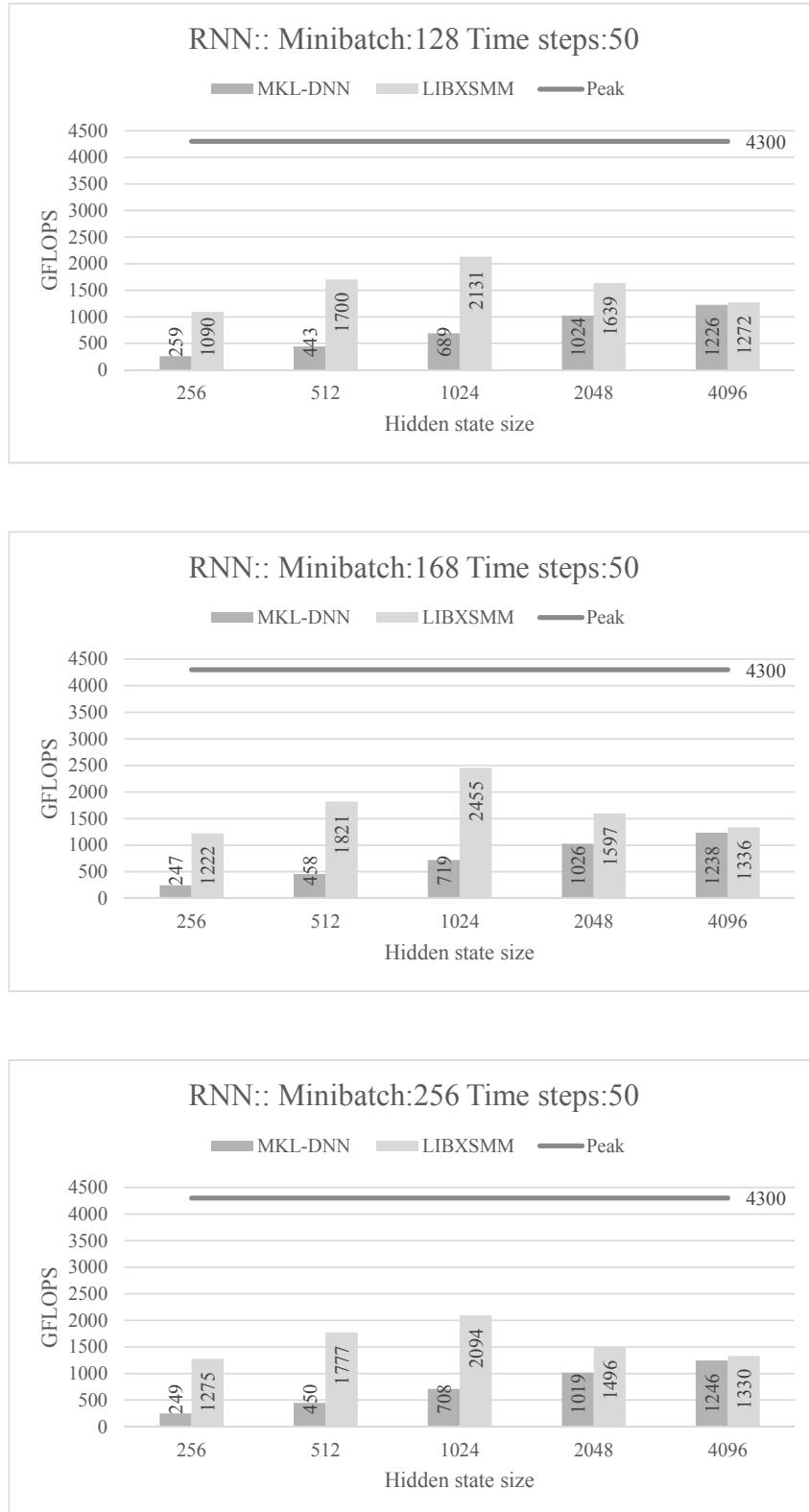
while it keeps the  $C$  accumulator in registers. Once a block of GEMM is computed, we apply element-wise operations on it while hot in cache. It is worth noting that we use Intel AVX512 intrinsics for vectorization and Intel Short Vector Math Library (SVML) for fast tanh and sigmoid computations. Same optimization principles are applied to backward and weight-update passes as well. Furthermore, our RNN operators are thread-library agnostic (can use any of pthreads, OpenMP, C++ threads, Cilk, TBB, etc.) – thus enabling an easy integration with any choice of framework.

Our experiments show that LIBXSMM RNN cell-forward pass can outperform that of MKL-DNN by  $\sim 3\times$  for smaller hidden-state sizes. However, this performance’s speed-up gradually decreases for larger hidden state sizes because GEMM has cubic complexity while the element-wise operations are quadratic and, as such for large sizes, the element-wise operations’ bandwidth overheads are less emphasized. The results are shown in Fig. 6. Note that we have reported the floor values of all numbers for better readability. The performance difference is even more highlighted for combined backward and weight update passes as given in Fig. 7. As can be seen in this figure, LIBXSMM RNN cell outperforms that of MKL-DNN for smaller hidden states by

up to  $\sim 5\times$ . It is worth noting that the activation function used in Fig. 6 and Fig. 7 is tanh; we see similar patterns for sigmoid and ReLU activation functions as well.



**Figure 6.** RNN cell-forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256



**Figure 7.** RNN cell-backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

### 3.1.2. LSTM cell efficiency

In LSTM, there are four different weights and recurrent weights; consequently, we block these in the following way:

- weights:  $[C][4K] \rightarrow [4K/B_K][C/B_C][B_C][B_K]$ ;
- recurrent weights:  $[K][4K] \rightarrow [4K/B_K][K/B_K][B_K][B_K]$ .

We compared our optimized LSTM cell with the latest MKL-DNN LSTM cell on a single socket Xeon Platinum 8280. The LIBXSMM LSTM cell-forward pass based on our approach always outperformed that of MKL-DNN and BLAS as shown in Fig. 10. For small/medium-size problems our implementation is  $\sim 1.3\times$  faster than MKL-DNN; however, with an increase in hidden-state size, the performance of MKL-DNN becomes comparable with ours. It is worth noting that since the GEMM sizes of LSTM are four times larger than RNN for given  $N$ ,  $C$  and  $K$  values, the blocking factors are accordingly smaller for LSTM, so that identical number of elements (as RNN) can fit into the cache. For the fused backward/weight-update pass, the LIBXSMM LSTM cell's performance is  $\sim 2.2\times$ , on average, with respect to MKL-DNN LSTM cell as shown in Fig. 11.

### 3.1.3. GRU cell efficiency

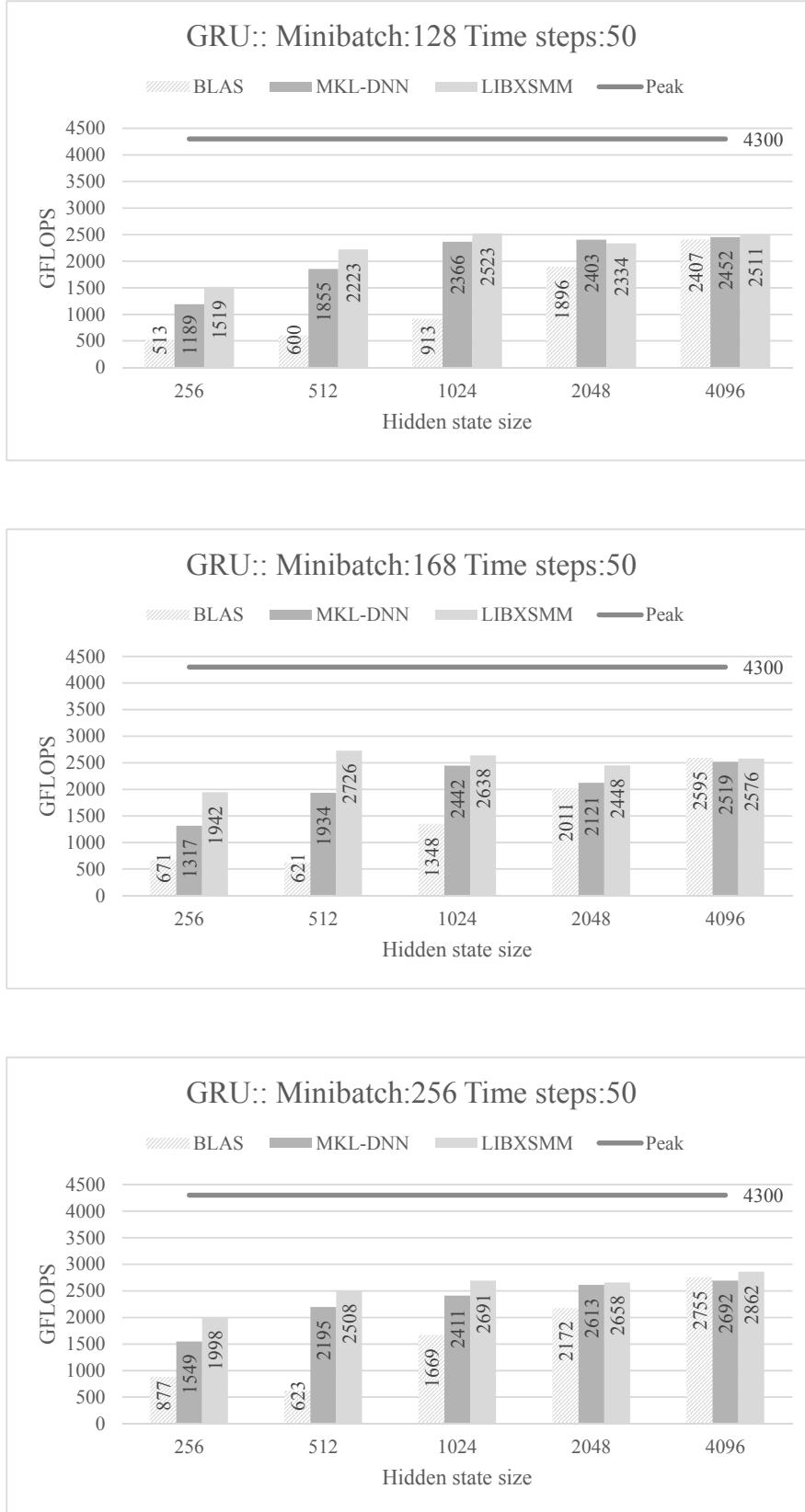
For GRU cell, which has three different weights and recurrent weights (in contrast to four of LSTM), we follow an identical policy of blocking the inputs and weights as that of LSTM. Hence, we do not elaborate on this cell. The results of its forward and backward/weight-update passes can be found in Fig. 8 and Fig. 9, respectively, where we can see that while the forward pass can be up to  $\sim 1.5\times$  faster, backward/weight-update pass can even surpass  $2\times$  speedup in comparison to MKL-DNN.

We believe a dataflow approach is better suited for CPUs than GPUs since it relies on coarse-grained parallelization and precise locality control. It is noteworthy that in LIBXSMM, the same optimizations as mentioned here have also been applied to the implementation of fully connected layer, which is the main computational kernel in multi-layer perceptron networks.

### 3.1.4. Application level impact of LSTM cell

Googles neural machine translation (GNMT) [24] is state-of-the-art LSTM-based language translation application. As shown in Fig. 12, we compare between four variants of GNMT code to highlight the speed-up that our code achieves for 8-layer German-to-English GNMT model. For all the experiments, we consider a minibatch size of 168 with the number of inter\_op\_threads equal to 1 and a number of intra\_op\_threads equal to 28.

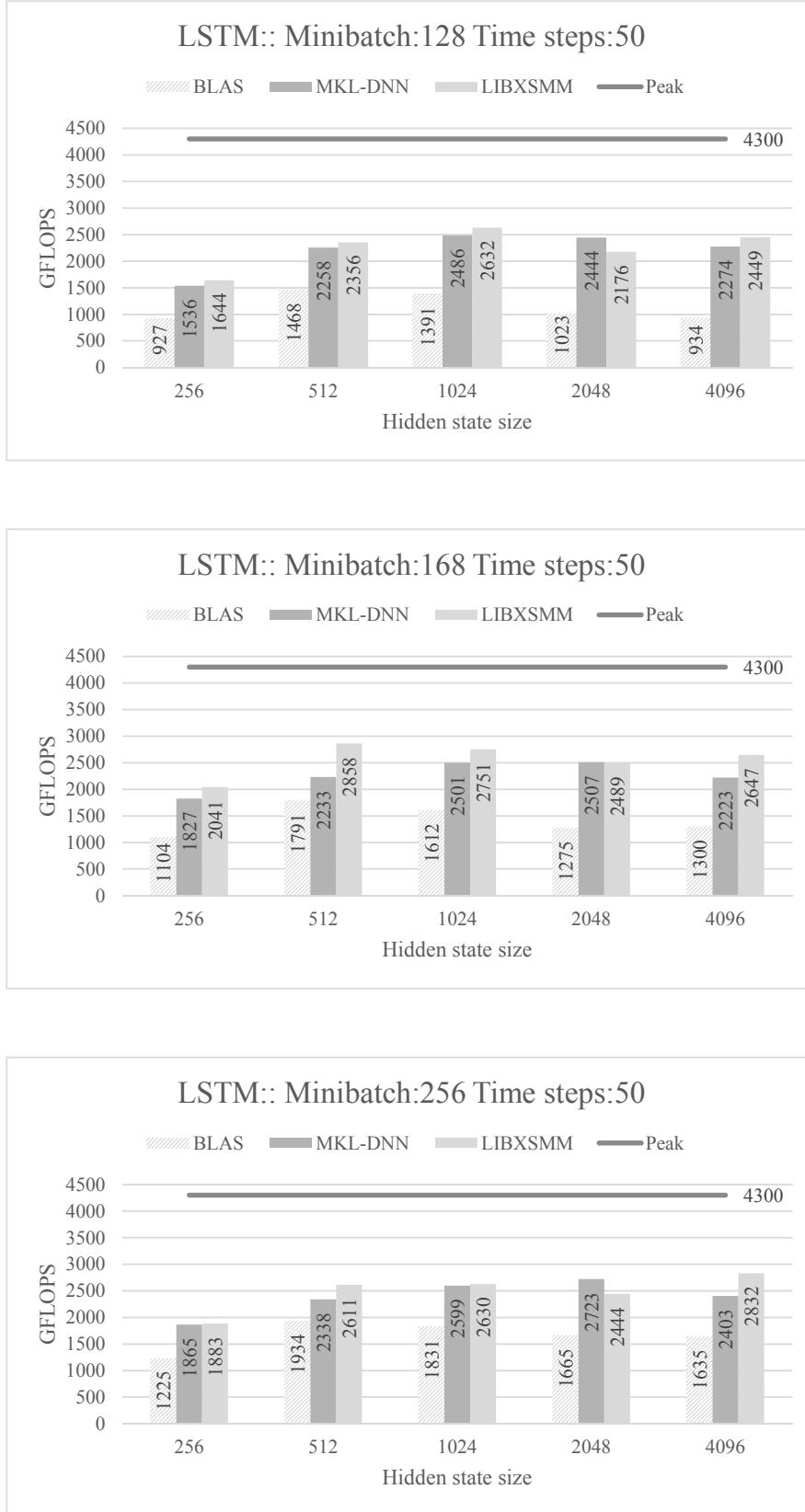
1. **Reference w/o MKL:** This is the default TensorFlow code which does not support MKL.
2. **Reference w/ MKL:** This is TensorFlow with MKL support.
3. **XsmmLSTM:** This is the GNMT code which has LIBXSMM LSTM cell integrated without the support for fused time steps. This is an intermediate code that we tried out because it allowed an easy integration with other TensorFlow wrappers, thereby enabling fast deployment. Moreover, we maintained the activation and the weight layouts identical to that of TensorFlow which aided in correctness checks.
4. **+Fused Encoder:** This code supports LIBXSMM LSTM cell having fused time steps along with optimal blockings for activations and weights. However, note that the LIBXSMM LSTM cells are deployed only for the encoders. We could not change decoders to use our cell because the time step loop for decoding stage is implemented inside seq2seq library [4] and we presently leave it as a future work. Note that at this step, we achieve  $2.35\times$  performance compared to the default implementation.



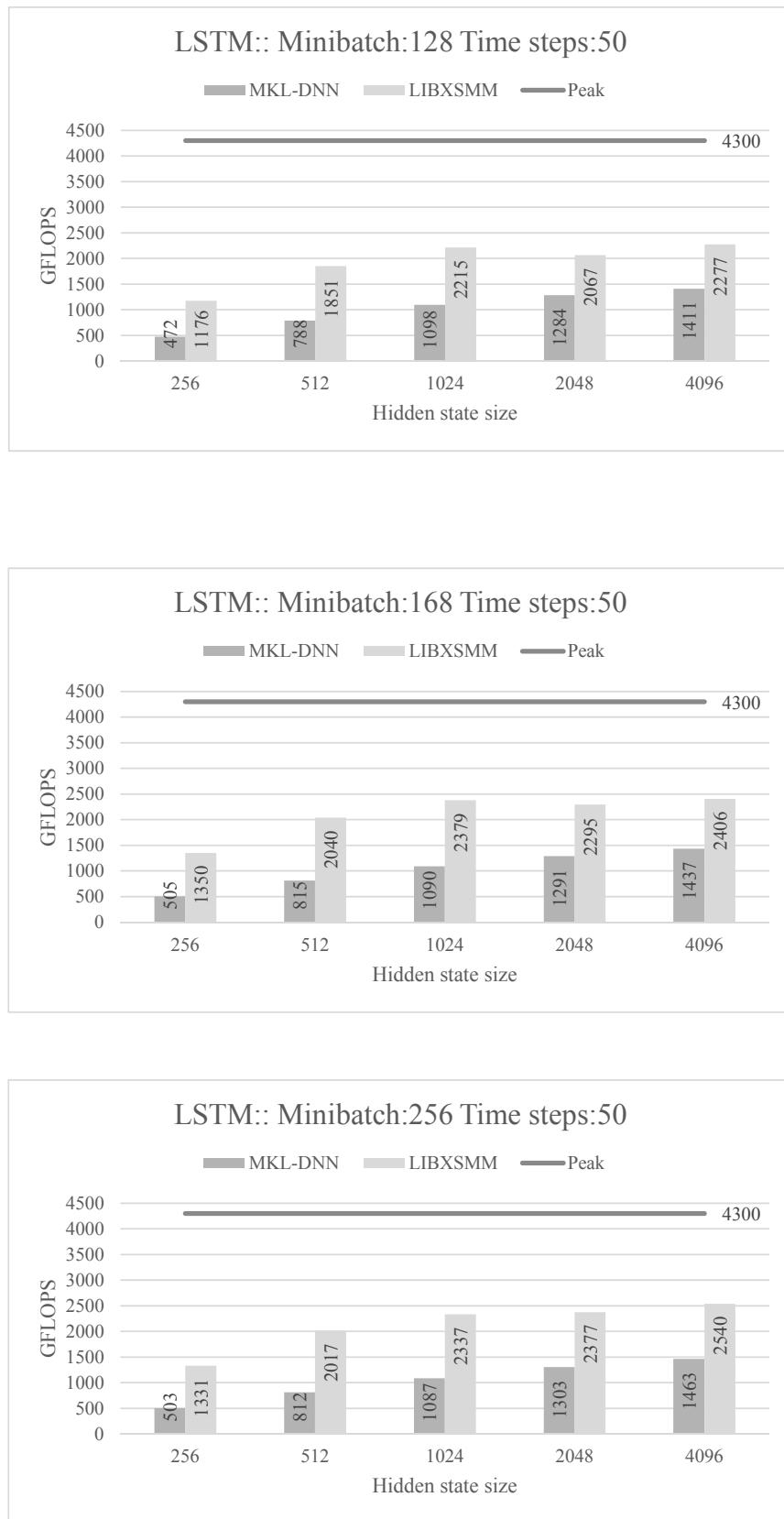
**Figure 8.** GRU cell forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256



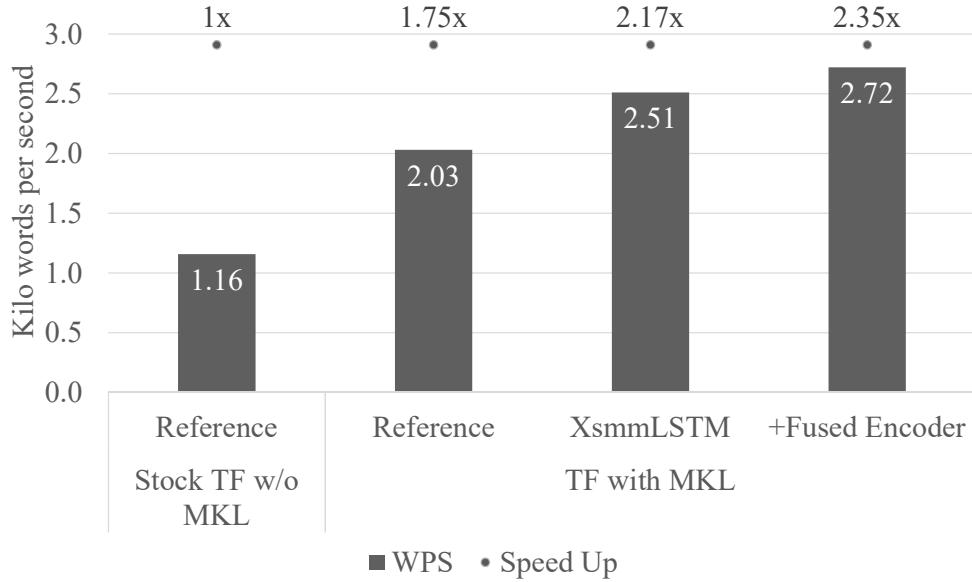
**Figure 9.** GRU cell backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256



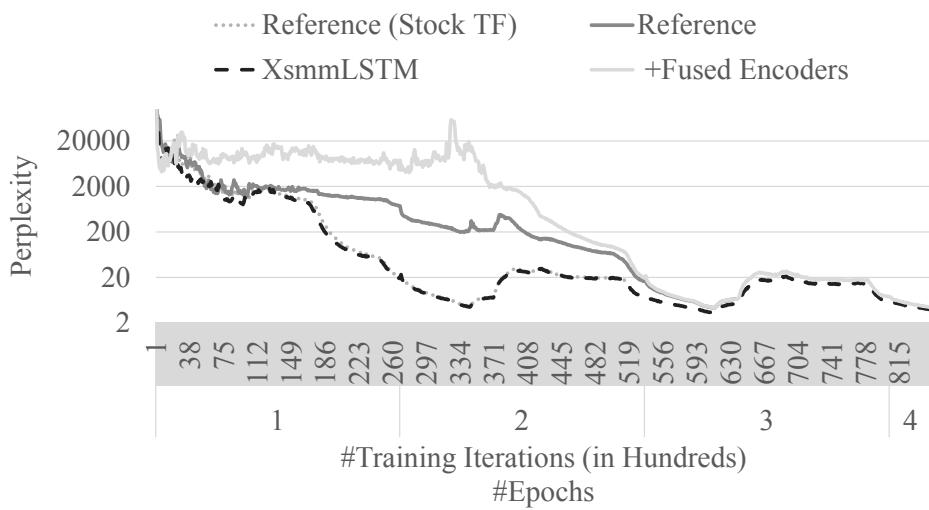
**Figure 10.** LSTM cell forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256



**Figure 11.** LSTM cell backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256



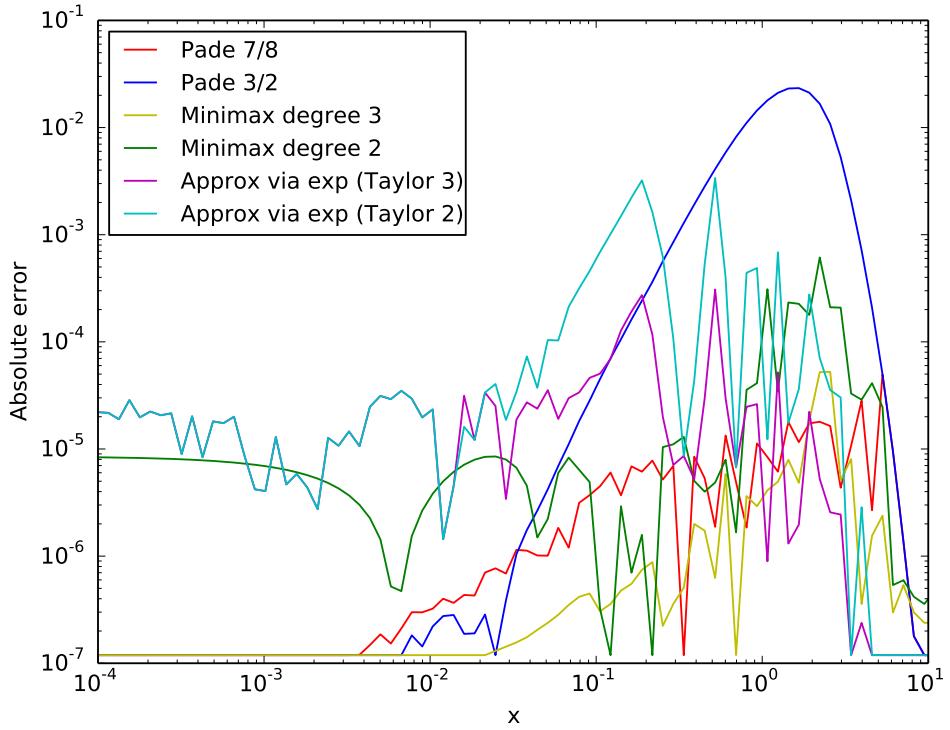
**Figure 12.** GNMT 4-layer performance on CascadeLake (with turbo enabled)



**Figure 13.** GNMT convergence: Perplexity

*Perplexity* is a measure of how easy a probability distribution is to predict; thus, the lower the value of perplexity, the better it is. Figure 13 shows how the perplexity varies during the training of the 8-layer GNMT model for the four different GNMT codes described above. As is evident from the figure, all the codes converge and follow a similar trend. In our recent work [17], we explain in detail how we have integrated our LIBXSMM LSTM cell (with fused time steps) into TensorFlow [5] framework and for the first time scaled GNMT to a 16-node Intel CPU cluster. Our code outperformed Googles stock CPU-based GNMT implementation by more than 25 $\times$  on the 16 node CPU cluster. Along with the efficient scaling libraries and a smart batching strategy, the LIBXSMM LSTM cell played a crucial role in obtaining this milestone.

### 3.2. Evaluation of Approximate Sigmoidal Functions



**Figure 14.** Absolute error of the tanh implementations on CascadeLake

In order to assess the accuracy and the performance of our approximations, we conducted experiments on Intel’s CascadeLake (CLX) as described earlier.

#### 3.2.1. Accuracy of approximations

Figure 14 (Left) shows the absolute error of the approximations on CLX in the positive range  $[10^{-4}, 10]$  since tanh is symmetric with respect to the origin. In regard to the rational approximations, the  $\text{Padé}_{[7/8]}$  has maximum absolute error of  $10^{-4}$ , whereas the  $\text{Padé}_{[3/2]}$  has maximum absolute error of  $10^{-2}$ . The  $2^{nd}$ - and  $3^{rd}$ -degree piecewise minimax polynomials have maximum absolute errors of  $10^{-3}$  and  $10^{-4}$ , respectively. Notably, the  $3^{rd}$ -degree minimax polynomials exhibit a smaller absolute error than the Padé rational approximations in most of this regime. Finally, regarding the tanh via exp() approximation with Taylor polynomials of order 2 and 3, we get maximum absolute errors of  $10^{-3}$  and  $10^{-4}$  respectively. We also observe that the latter approximations are more accurate among all other considered variants for large  $x$  values. These observations imply that one can get an even better approximation by combining properly the aforementioned algorithms in the relevant intervals, similarly to previous work [6]. In this work, since we prioritize the speed of the approximations, we do not consider such hybrid algorithms.

At first sight these errors are undoubtedly much higher than single precision’s machine epsilon of roughly  $10^{-7}$ . However, one has to keep in mind that the emerging datatypes for deep learning training are float16 which offers a bit more than three digits precision and bfloat16 which has a bit less than three digits of precision. Deep learning inference tasks are often able to run fine with 8 bit fix-point datatypes. Given such input data, the obtained accuracies are perfectly sufficient as they are in the regime of these datatypes’ machine epsilons. Figure 14 shows the absolute error of the approximations on CLX.

**Table 1.** Performance of the various implementations on CascadeLake

Algorithm	Cycles per tanh computation
Rational Padé 3/2	0.39
Rational Padé 7/8	0.59
Minimax polynomials of degree 2	0.35
Minimax polynomials of degree 3	0.42
Approximation via exp (Taylor degree 2)	0.42
Approximation via exp (Taylor degree 3)	0.47
SVML (high precision)	1.53
SVML (low precision)	0.95
libm	28.32

### 3.2.2. Performance evaluation

Table 1 shows the performance of various approximation algorithms on CLX, where the figure of merit is cycles-per-tanh computation (since all implementations leverage AVX512 instructions, each function call performs 16 tanh computations at once). In addition to our implementations, we measured the performance of the Intel Short Vector Math Library (SVML) and the tanh implementation in libm.

On CLX, the fastest implementations are the 2<sup>nd</sup>-degree minimax polynomials and the Padé[3/2] approximation with 0.35 and 0.39 cycles-per-tanh computation, respectively. Among these two implementations and given the absolute errors depicted in Fig. 14, we conclude that the 2<sup>nd</sup>-degree minimax polynomials yield the best tradeoff in speed and accuracy. This approximation is 2.7× faster than the low-precision SVML tanh implementation and 81× faster than the implementation in libm. The accuracy level of the 2<sup>nd</sup>-degree minimax polynomials should be sufficient for most of the low-precision deep learning applications. However, if accuracy higher than the Padé[7/8] requested, the 3<sup>rd</sup>-degree minimax polynomials and the tanh via exp() approximation with Taylor polynomial of order 3 may be used, yielding speedups of 1.6×, 2.3× and 2× over the low-precision SVML tanh implementation.

We further used the Intel Architecture Code Analyzer (version v3.0-28) to assess how the instructions are scheduled on CLX. As an illustrative example, we will use the assembly code of the benchmarking loop with the 3<sup>rd</sup>-degree minimax polynomials exhibited at Tab. 2. Based on the scheduling of the instructions, we observe that the critical path is determined by ports P0 and P5 where the permute, FMAs and the remaining vector compute instructions are scheduled. This critical path with the length of 6.5 cycles determines the reciprocal throughput, and this calculation agrees with our empirical result: each iteration computes 16 tanh evaluations; therefore, each tanh evaluation is estimated to take 6.5/16 = 0.41 cycles, which is close to the measured throughput 0.47.

Finally, by considering the instruction mix in the approximations, we gain some intuition about how to create hardware that accelerates such implementations. For example, a common method used in all our approximation algorithms is the polynomial evaluation via FMA instructions that implement the Horner’s rule. One could accelerate such method via fixed function hardware, e.g. by chaining  $k$  FMA units to perform a  $k$ -th degree polynomial evaluation. Such specialized hardware can provide faster and higher accuracy approximations by making high degree polynomial computations inexpensive.

**Table 2.** Assembly code and CLX instruction scheduling of the benchmarking loop with the 3<sup>rd</sup> degree minimax polynomials

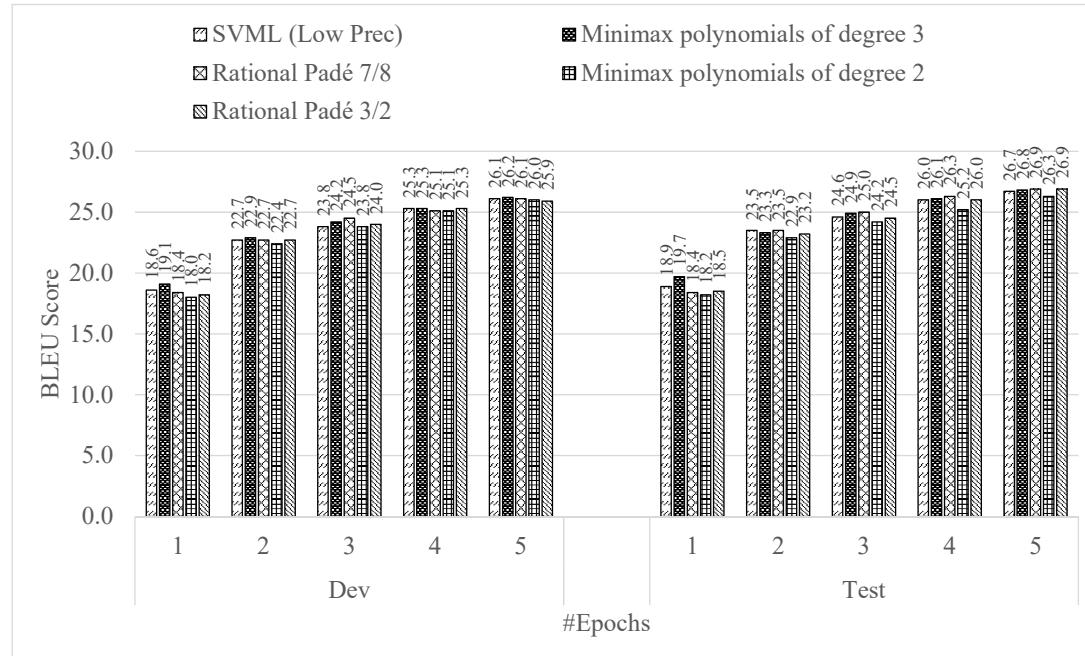
Uops	Ports pressure in cycles							Instructions
	P0	P1	P2	P3	P4	P5	P6	
1						1.0		inc eax
1			1.0					vmovups zmm10, zmmword ptr [rsp+0x600]
1	0.5				0.5			vandps zmm0, zmm10, zmm8
1	0.5				0.5			vandps zmm1, zmm10, zmm9
1	1.0							vpsrld zmm11, zmm0, 0x16
1	1.0							vpmmaxsd zmm12, zmm11, zmm7
1	1.0							vpmminsd zmm13, zmm12, zmm6
1					1.0			vpermps zmm15, zmm13, zmm3
1					1.0			vpermps zmm14, zmm13, zmm2
1					1.0			vpermps zmm10, zmm13, zmm4
1					1.0			vpermps zmm11, zmm13, zmm5
1	1.0							vfmadd231ps zmm15, zmm14, zmm0
1	0.5				0.5			vfmadd231ps zmm10, zmm15, zmm0
1	0.5				0.5			vfmadd213ps zmm0, zmm10, zmm11
1	0.5				0.5			vxorps zmm0, zmm0, zmm1
2				1.0	1.0			vmovups zmmword ptr [rsp+0x640], zmm0
1								cmp eax, r13d
0								jl 0xffffffffffff94

### 3.2.3. Application level impact of approximate tanh

In order to study the impact of approximate tanh at the application level, we implemented LSTM cell with various approximate tanh implementations and used these LSTM cells for Google Neural Machine Translation (GNMT) training. We ran our convergence experiments on a small 4-node CLX cluster with global minibatch size of 1024 for German-to-English WMT16 training dataset using Adam solver with learning rate of 0.0005 for 5 epochs. We evaluated translation accuracy as a BLEU score after every epoch using newstest2013 as development (DEV) and newstest2015 as test (TEST) datasets. As seen in the Fig. 15, there is little impact of tanh approximation on training convergence and even though there is some variation at the beginning of the training, at the end of 5<sup>th</sup> epoch, all the evaluated versions converge to similar accuracy. Particularly, Rational Padé approximation produces best TEST BLEU score of 26.9 which is slightly better than the BLEU score of 26.7 for the reference SVM version. Similarly, a minimax polynomial-degree 3 version produces the best DEV score of 26.2 compared to 26.1 for the reference SVM version.

## Conclusion

In summary, we propose an implementation of LSTM cell using a “dataflow”-approach small-blocked GEMMs instead of large GEMMs on Intel Xeon architecture. Our strategy helps in maximizing locality, weight reuse and fuse element-wise operations which are, otherwise, exposed as bandwidth-bound kernels. For small/medium sized problems, our implementation of RNN-forward pass is  $\sim 3\times$  faster than the MKL-DNN cell, while for backward/weight update it is up to  $\sim 5\times$  faster. For large weight matrices, the two approaches, however, have similar performance which stems from the fact that GEMM has cubic complexity, whereas fusing element-wise



**Figure 15.** Convergence of GNMT with various approximate tanh implementations

operations are only of quadratic complexity. It should be noted that this conclusion might change with GEMM acceleration hardware.

We also assessed the accuracy/speed tradeoffs of the various implementations of sigmoid and tanh functions on Intel’s CascadeLake processor. Our approximations obtain accuracies that are sufficient for contemporary inference and training deep learning applications with low precision (e.g. float16, bfloat16, 8 bit fix-point datatypes) while they outperform the low-precision tanh SVML implementations by factors of 1.6–2.6×. We envision that our implementations will have a magnified importance in the context of emerging deep-learning hardware that accelerates GEMM-flavored computations and, as a result, a fast evaluation of non-linear activation functions is necessitated.

*This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.*

## References

1. BFLOAT16 - hardware numerics definition. <https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf>, accessed: 2019-03-22
2. Intel(R) Math Kernel Library for Deep Neural Networks. <https://github.com/intel/mkl-dnn>, accessed: 2019-03-22
3. LIBXSMM. <https://github.com/hfp/libxsmm>, accessed: 2019-09-13
4. Module: tf.contrib.seq2seq. [https://www.tensorflow.org/api\\_docs/python/tf/contrib/seq2seq](https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq), accessed: 2019-04-08

5. Abadi, M., Barham, P., Chen, J., *et al.*: Tensorflow: A system for large-scale machine learning. In: OSDI. pp. 265–283 (2016)
6. Beebe, N.H.: Accurate hyperbolic tangent computation. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah (1991)
7. Brezinski, C.: Outlines of padé approximation. In: Computational aspects of complex analysis, pp. 1–50. Springer (1983)
8. Chen, M., Ding, G., Zhao, S., *et al.*: Reference based LSTM for image captioning. In: AAAI. pp. 3981–3987 (2017)
9. Chetlur, S., Woolley, C., Vandermersch, P., *et al.*: cuDNN: Efficient primitives for deep learning. CoRR abs/1410.0759 (2014), <http://arxiv.org/abs/1410.0759>
10. Chung, J., Gülcühre, Ç., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR abs/1412.3555 (2014), <http://arxiv.org/abs/1412.3555>
11. Cody, W.J.: Software Manual for the Elementary Functions (Prentice-Hall series in computational mathematics). Prentice-Hall, Inc. (1980)
12. Courbariaux, M., Bengio, Y., David, J.P.: Training deep neural networks with low precision multiplications. arXiv preprint arXiv:1412.7024 (2014)
13. Das, D., Mellempudi, N., Mudigere, D., *et al.*: Mixed precision training of convolutional neural networks using integer operations. arXiv preprint arXiv:1802.00930 (2018)
14. Elsen, E.: Optimizing rnn performance. [http://svail.github.io/rnn\\_perf/](http://svail.github.io/rnn_perf/), accessed: 2019-03-28
15. Graves, A., Liwicki, M., Fernandez, S., *et al.*: A novel connectionist system for unconstrained handwriting recognition. IEEE Trans. Pattern Anal. Mach. Intell. 31(5), 855–868 (2009), DOI: 10.1109/TPAMI.2008.137
16. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation 9(8), 1735–1780 (1997), DOI: 10.1162/neco.1997.9.8.1735
17. Kalamkar, D., Banerjee, K., Srinivasan, S., *et al.*: Training google neural machine translation on an intel cpu cluster. In: CLUSTER (2019 (to appear))
18. Karpathy, A.: The unreasonable effectiveness of recurrent neural networks. <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>, accessed: 2019-03-28
19. Namin, A.H., Leboeuf, K., Muscedere, R., Wu, H., Ahmadi, M.: Efficient hardware implementation of the hyperbolic tangent sigmoid function. In: ISCAS. pp. 2117–2120 (2009), DOI: 10.1109/ISCAS.2009.5118213
20. Powell, M.J.D.: Approximation theory and methods. Cambridge university press (1981)
21. Rivlin, T.J.: The Chebyshev polynomials (Pure and Applied Mathematics). Wiley-Interscience (1974)

22. Tommiska, M.T.: Efficient digital implementation of the sigmoid function for reprogrammable logic. IEE Proceedings - Computers and Digital Techniques 150(6), 403–411 (2003), DOI: 10.1049/ip-cdt:20030965
23. Wen, T., Gasic, M., Mrksic, N., *et al.*: Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In: EMNLP. pp. 1711–1721 (2015), DOI: 10.18653/v1/D15-1199
24. Wu, Y., Schuster, M., Chen, Z., *et al.*: Google’s neural machine translation system: Bridging the gap between human and machine translation. CoRR abs/1609.08144 (2016), <http://arxiv.org/abs/1609.08144>

# A Skewed Multi-banked Cache for Many-core Vector Processors\*

*Hikaru Takayashiki<sup>1</sup> , Masayuki Sato<sup>1</sup> , Kazuhiko Komatsu<sup>1</sup> ,*  
*Hiroaki Kobayashi<sup>1</sup> *

© The Authors 2019. This paper is published with open access at SuperFri.org

As the number of cores and the memory bandwidth have increased in a balanced fashion, modern vector processors achieve high sustained performances, especially in memory-intensive applications in the fields of science and engineering. However, it is difficult to significantly increase the off-chip memory bandwidth owing to the limitation of the number of input/output pins integrated on a single chip. Under the circumstances, modern vector processors have adopted a shared cache to realize a high sustained memory bandwidth. The shared cache can effectively reduce the pressure to the off-chip memory bandwidth by keeping reusable data that multiple vector cores require. However, as the number of vector cores sharing a cache increases, more different blocks requested from multiple cores simultaneously use the same set. As a result, conflict misses caused by these blocks degrade the performance.

In order to avoid increasing the conflict misses in the case of the increasing number of cores, this paper proposes a skewed cache for many-core vector processors. The skewed cache prevents the simultaneously requested blocks from being stored into the same set. This paper discusses how the most important two features of the skewed cache should be implemented in modern vector processors: hashing function and replacement policy. The proposed cache adopts the odd-multiplier displacement hashing for effective skewing and the static re-reference interval prediction policy for reasonable replacing. The evaluation results show that the proposed cache significantly improves the performance of a many-core vector processor by eliminating conflict misses.

*Keywords:* HPC, vector architecture, cache, skewed-associativity.

## Introduction

Modern vector processors achieve high computing capability and high memory performance. Enhancing the architecture specialized for vector instructions with long vector lengths and the memory system focusing on high memory bandwidth, modern vector processors can achieve high sustained performances in scientific and engineering applications. The performance requirement of these applications is growing because these applications are eagerly developed for the demands of improving their accuracy and widening the applicable area. Thus, vector processors have been expected to provide higher performance.

In order to provide higher performance, modern vector processors have adopted two features different from the typical one. First, modern vector processors have increased their computing capability by increasing the number of vector cores, even though historically the vector processors have a single powerful core. As this trend may continue, many-core technology will play an essential role in driving computing capability growth in the future. Second, modern vector processors adopt multi-banked caches to keep the high sustained memory bandwidth. Since the further improvement of off-chip memory bandwidth is difficult due to the limitation of the number of input/output pins integrated on a single chip, a cache provides reusable data to vector cores at the high bandwidth.

As the number of vector cores increases, the off-chip memory readily becomes a bottleneck on memory-intensive applications in the future. In order for the vector processors to mitigate the gap between computing capability and off-chip memory performance, efficient utilization of the shared cache becomes an essential key factor. Since the shared cache can reuse the data among the

---

\*The paper is recommended for publication by the Program Committee of the International Supercomputing Conference 2019 “HPC in Asia”.

<sup>1</sup>Tohoku University, Sendai, Japan

vector cores, unnecessary off-chip memory accesses can be reduced if the same data already exists on the shared cache. Thus, applications that run on the future vector processors are ought to be optimized to arrange the data on the shared cache as much as possible. Under the well-optimized applications, the number of cache hits will certainly increase if the number of vector cores sharing the data on the cache increases. Thus, less off-chip memory pressure can be realized by increasing the number of vector cores sharing data on the shared cache as much as possible.

First, this paper preliminarily evaluates the effect of the shared cache organizations on a many-core vector processor. A simple optimization is applied to an application so that more data can be shared as the number of vector cores increases. Thus, if the number of vector cores that share the cache architecturally increases, the more cache hit rate will be obtained. However, the preliminary evaluation clarified that a cache shared by many vector cores suffers from a lot of conflict misses. Although increasing the cache associativity is a common solution that can reduce the conflict misses, this solution brings a substantial overhead for a multi-banked cache.

Therefore, this paper proposes a skewed cache for many-core vector processors. The skewed cache is a cache that adopts skewed-associativity [13]. The skewed cache can suppress the number of conflict misses by preventing the simultaneous data requests of multiple vector cores from using the same cache set. This paper also discusses two features of the skewed cache: hashing functions, and replacement policies. The proposed cache adopts the odd-multiplier displacement hashing for effective skewing and the static re-reference interval prediction policy for reasonable replacing. In the evaluation, this paper evaluates the cache hit rates by using a stencil calculation kernel with varying the number of vector cores sharing a cache and its associativity. The evaluation results show that the proposed cache can eliminate the conflict misses and achieve nearly ideal hit rates in the case of the shared cache configurations.

The rest of the paper is organized as follows. Section 1 introduces a many-core vector processor assumed in this paper and indicates that the multi-banked cache suffers from conflict misses through the preliminary evaluation. Section 2 proposes a skewed multi-banked cache for many-core vector processor. Section 3 evaluates the proposal by simulation and discusses the results. The final section concludes this paper.

## 1. Challenges in Many-core Vector Processors

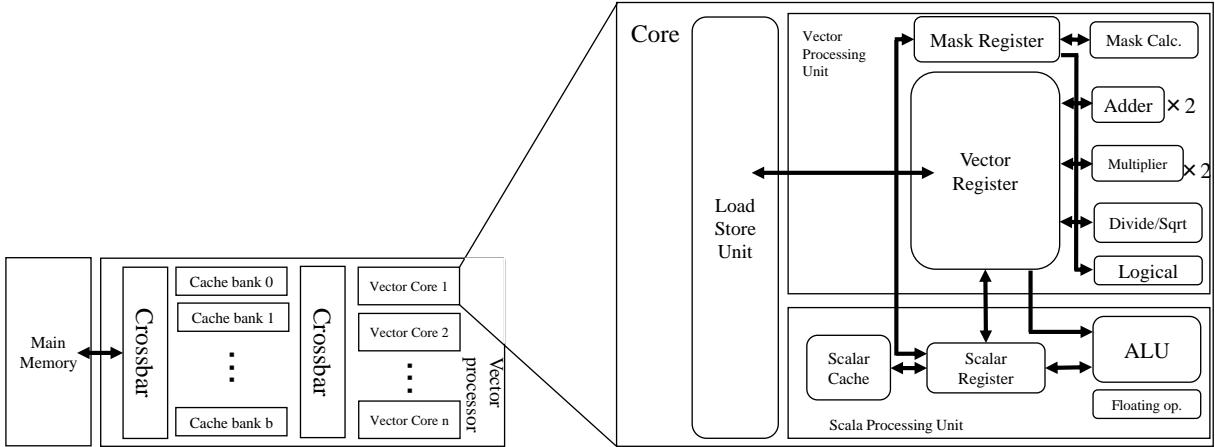
This section describes an organization of a many-core vector processor assumed in this paper. Then, various shared cache configurations of the many-core vector processor are preliminarily evaluated to show the impacts of conflict misses on the hit rates of the shared caches.

### 1.1. Many-core Vector Processors

#### 1.1.1. Vector cores

Figure 1 shows the many-core vector processor assumed in this paper. One of the main parts of this processor is a vector core. The vector core is mainly divided into a scalar processing unit, a vector processing unit, and a load/store unit. The scalar processing unit fetches and decodes all the instructions and is responsible for the subsequent execution stages of scalar instructions.

The vector processing unit consists of vector operating units and vector registers. The vector operating units execute vector instructions, which can apply the arithmetic operations to vectorized data. A vector instruction can operate multiple elements at once. The number of elements



**Figure 1.** The overview of the many-core vector processor

that can be operated by one vector instruction is called *vector length*. For example, the maximum vector length is 256 in double-precision floating-point elements in the latest vector processor, SX-Aurora TSUBASA [8]. Compared to SIMD instructions of general-purpose processors, vector instructions can process large amount of data by one instruction.

The vector registers store operands used for executing vector instructions. The vector functional units can always be fed by the elements from the vector registers for calculations. These units consist of multiple arithmetic pipelines of addition, multiplication, division/square root, and logical calculation. Each unit can operate independently. Thus, it is possible to perform multiple types of operations in parallel. The vector mask register masks the execution results of a vector instruction. By using the vector mask register, loops including conditional branches are executed by using vector instructions.

The load/store unit generates memory addresses from vector load/store instructions and continuously sends memory access requests to the memory system. After all the data have arrived from the memory system, the data are transferred to the vector register immediately.

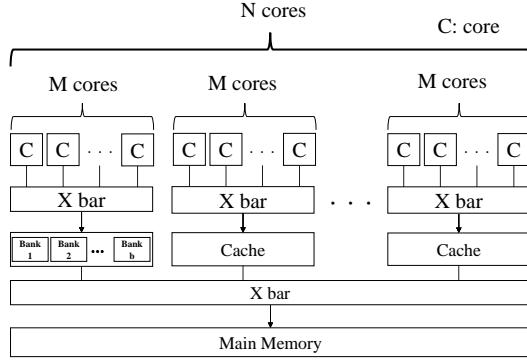
### 1.1.2. Multi-banked shared cache

With the improvement of the computing capability of vector cores, the demand for memory performance also increases to supply the data required by vector cores. However, the improvement of memory performance is behind in that of computing capability. Hence, cache memory is essential to fulfill the gap between them, and it is implemented in modern vector processors. For example, since the SX-9 or later vector processors of NEC SX series, an on-chip multi-banked cache memories are implemented inside the processor [5, 8]. This paper also assumes that the many-core vector processor includes a multi-banked cache memory.

If the number of vector cores increases in the future, it is considered that several numbers of cores are connected to one cache. Figure 2 shows an example where each cache is shared by  $M$  vector cores when the total number of vector cores is  $N$ . The following equation expresses the relationship between  $M$  and  $N$

$$C = \frac{N}{M}, \quad (1)$$

where  $C$  is the number of caches, and  $M$  means the number of vector cores connected to one cache. Equation (1) indicates that the number of caches  $C$  varies depending on the  $M$  and  $N$ .



**Figure 2.** Example of  $M$  cores sharing the same cache in the  $N$  vector cores processor

---

```

1: for  $z = 1, \dots, N_z - 1$  do
2:   for  $y = 1, \dots, N_y - 1$  do
3:     for  $x = 1, \dots, N_x - 1$  do
4:        $b[z][y][x] = (a[z][y][x] +$ 
5:            $a[z][y][x-1] + a[z][y][x+1] +$ 
6:            $a[z][y-1][x] + a[z][y+1][x] +$ 
7:            $a[z-1][y][x] + a[z+1][y][x]) / 7$ 
8:     end for
9:   end for
10: end for

```

---

**Figure 3.** 3D 7-point stencil calculation

Thus, the number of cache banks in a cache,  $b$ , is expressed by the following equation.

$$b = \frac{B}{C} = \frac{BM}{N}, \quad (2)$$

where  $B$  is the total number of banks in a many-core vector processor.

In this paper,  $N$  and  $B$  are constant because the various configurations of the shared cache are examined under the same computing capability and memory performance. Thus, the number of vector cores connected to the cache  $M$  only changes the number of banks per cache  $b$  according to Equation (2). Moreover, the capacity of each bank is fixed so that the number of banks per cache determines the cache capacity.

## 1.2. Conflict Misses on The Many-core Vector Processor

### 1.2.1. 3D 7-point stencil calculation

This paper examines various configurations of the shared cache using a representative memory-intensive computing kernel, the 3D 7-point stencil kernel. The stencil computations including this kernel occupy major roles in scientific and engineering simulation codes.

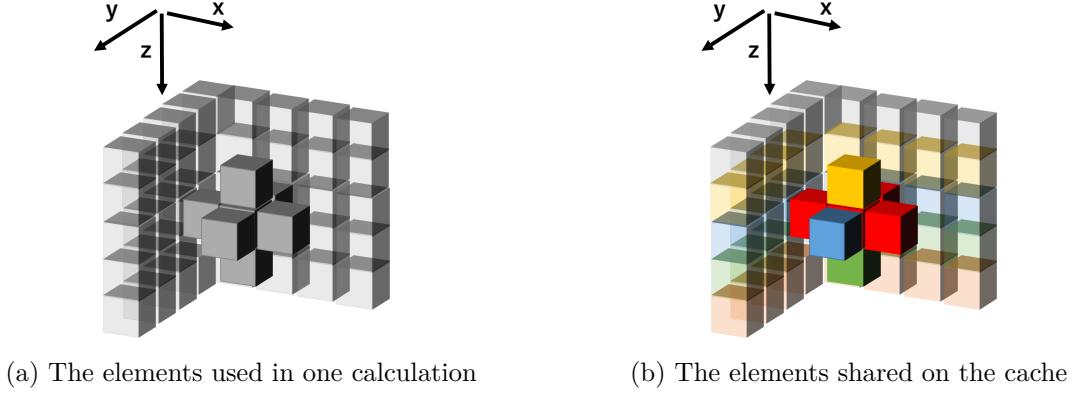
Figure 3 shows a pseudo-code for the 3D 7-point stencil kernel. This code derives arithmetic mean of a central element and its six neighboring elements along with  $x$ ,  $y$ , and  $z$  axes. The calculation is repeated for all the elements in the 3D space. Thus, for each iteration of the innermost loop of Fig. 3, a total of seven elements are required.

Figure 4(a) shows the elements used for one calculation in Fig. 3. The translucent elements represent the calculation space, and solid elements indicate the seven elements used for the calculation.

### 1.2.2. Stencil calculation with a shared cache

This paper assumes that the parallelization is performed based on the outermost loop regarding  $z$ -axis in Fig. 3. Therefore, each  $z$ -axis iteration of the loop is cyclically assigned to each core. The  $k$ -th iteration of the outermost loop is calculated by the core whose ID is  $(k \bmod N)$ .

In the stencil calculation, each vector core accesses the central element and its neighboring elements. If other vector cores are already accessing these neighboring elements, the data might



**Figure 4.** 3D 7-point stencil kernel calculation

have already been placed in the cache. Thus, these elements can be reused on the cache, resulting in pressure reduction to off-chip memory.

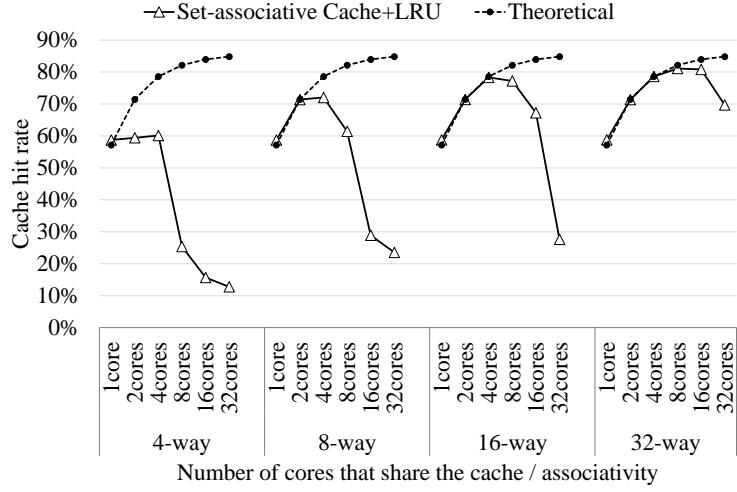
Based on this parallelization, the theoretical cache hit rate of the 3D 7-point stencil calculation can be derived. Note that, in this paper, one layer of elements in an x-y plane cannot fit in the cache while one line of elements along with the y-axis can. First, the theoretical cache hit rate is calculated in the case where the vector core does not share the cache at all. Figure 4(b) shows which elements will hit on the cache. The translucent elements show the group of the elements that are calculated by one core. The solid blue element in Fig. 4(b) should miss irrespective of whether the cache is shared or not. The red elements:  $a[z][y][x]$ ,  $a[z][y][x-1]$ ,  $a[z][y][x+1]$ , and  $a[z][y-1][x]$  should hit because the last iteration brings those elements in the cache irrespective of whether the cache is shared. The solid yellow and green elements cannot become hits because the one layer of the x-y plate cannot fit in the cache. Thus, in the case where the vector cores do not share a cache, the theoretical hit rate becomes 4/7 (57.14 %).

Next, the theoretical cache hit rate is calculated in the case where the vector cores share the cache with its neighboring cores, as shown in Fig. 2. The yellow and green elements can hit by sharing the cache because neighboring cores also bring the elements in the cache. Thus,  $a[z-1][y][x]$  and  $a[z+1][y][x]$  can hit additionally. However, since there is only one neighboring core for the cores which core numbers are 0 or  $M - 1$ , either  $a[z-1][y][x]$  or  $a[z+1][y][x]$  always misses for these cores. From this observation, the 3D 7-point stencil calculation can be shared at most 2/7 (28.57 %) of elements by the shared cache.

Here, there are  $7 \times M$  accesses by  $M$  cores when  $M$  iterations of the outermost loop are calculated in parallel. If the first core is calculating the outermost loop 1, the second core is calculating loop 2, ..., the  $M$ -th core is calculating loop  $M$ ; the core in charge of loop 1 and the core in charge of loop  $M$  can reuse only 5 elements by the shared cache, and the other cores can reuse 6 elements by the shared cache. From this observation, for all the cores, the number of hits in one iteration is calculated as  $6(M - 2) + 5 \times 2$ . Thus, the theoretical hit rate of 3D 7-point stencil calculation can be expressed as the following equation

$$H_7 = \frac{6(M - 2) + 5 \times 2}{7M} = \frac{6}{7} - \frac{2}{7M}. \quad (3)$$

From Equation (3), it is possible to predict the hit rate of the stencil calculation theoretically from the number of cores sharing one cache  $M$ . It can be seen that, as the number of cores  $M$  increases, the cache hit rate monotonically increases, asymptotically approaching to 6/7 (85.71 %).



**Figure 5.** The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed

Overall, increasing the number of vector cores sharing a cache enables more vector cores to reuse the data on the shared cache. In the stencil calculation, the theoretical cache hit rate can increase if more cores share a cache.

### 1.3. Preliminary Evaluation

Equation (3) expresses the upper-bound of the cache hit rate because this equation only considers the reusability of the elements, and other effects are ignored. In order to confirm the model defined by Equation (3), a preliminary evaluation is conducted to investigate the hit rate in the many-core vector processor by varying the number of cores that share a cache. The stencil calculation is parallelized to share the data, as discussed in Section 1.2.2. Thus, the larger number of cores suggests the higher cache hit rate, as expected from Equation (3). Under this situation, various configurations of a many-core vector processor are evaluated. The number of cores sharing a cache is set to 1, 2, 4, 8, 16, and 32, and the cache associativity is set to 4, 8, 16, and 32. The other configurations correspond to the configuration described in Section 3.1.

Figure 5 shows the results of the preliminary evaluation. The vertical axis indicates the cache hit rate, and the horizontal axis indicates the number of cores sharing one cache and associativity. In Fig. 5, the theoretical cache hit rate increases as increases the number of cores sharing one cache. On the other hand, the cache hit rate of the set-associative cache with LRU replacement policy becomes significantly low in the case of a large number of cores sharing one cache and the low associativity. This is because, when the stencil calculation is executed in parallel, multiple cores simultaneously access the same set, resulting in conflict misses. Therefore, the number of conflict misses must be reduced to exploit the effect of sharing data among the vector cores by the shared cache.

### 1.4. Related Work

One of the methods to reduce conflict misses is to increase the associativity; however, it is challenging to realize the higher associativity because of cost, power consumption, and area overheads on the chip in large-capacity and multi-banked caches. Therefore, this paper discusses the other ways to eliminate conflict misses instead of increasing the associativity.

Some papers have tackled with the reduction in the number of conflict misses. Qureshi *et al.* [11] have proposed the V-Way cache, which includes the flexible tag mechanism, enables a global replacement to eliminate conflict misses. The V-Way cache has additional tag entries so that associativity of a set can vary depending on the demand to the set. Incidentally, the tag and data entries are associated with each other by indirection pointers. This mechanism enables to select victims flexibly from the global view of data, not limited to the same set. However, the V-Way cache requires a lot of additional hardware cost for the implementation.

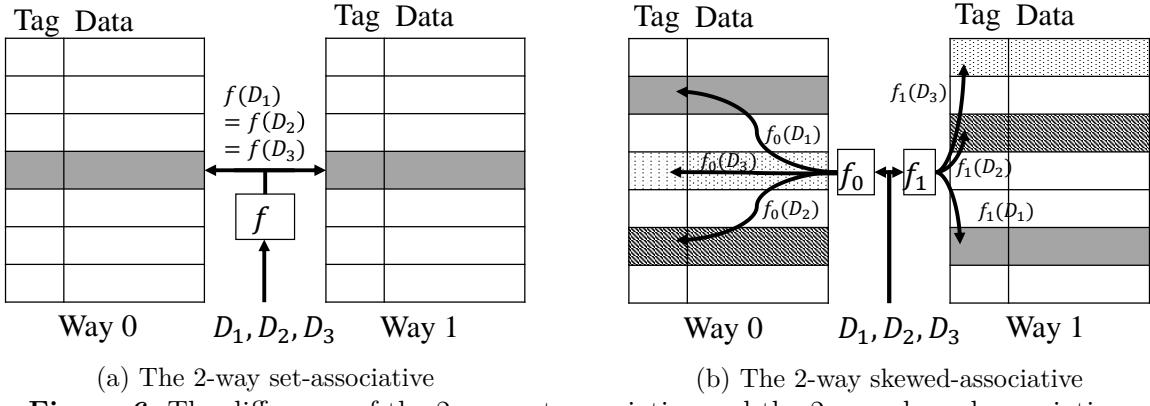
Sanchez *et al.* [12] have proposed the ZCache. The ZCache has focused on an insight that conflict misses occur by the lack of replacement candidates on an eviction. Hence, the ZCache selects several replacement candidates using multiple hashing functions for the same way on each miss. If a necessary block is about to be evicted, another useless block of another way would be evicted so that the necessary block can be relocated to and kept in the cache. However, the ZCache requires several array lookups to find a block in the cache, and additional data movement for the relocations. Consequently, these features cause too much cost for the multi-banked cache focusing on high bandwidth.

Although these proposals are effective in eliminating conflict misses, they require unignorable hardware and performance overheads. Moreover, the effect of these proposals is only evaluated for the scalar processors, not for the vector processors. Therefore, this paper focuses on a more straightforward way to reduce the number of conflict misses at low costs for the vector processors, looking back to the beginning of skewed-associativity.

There are several studies to clarify the usefulness of caches in vector processors. Musa *et al.* [10] have clarified that a four-core vector processor with one shared cache or two shared caches can improve the performance by 15-40% compared to the case without caches. This is because neighboring cores can reuse data by sharing the cache. Thus, the cache hit rate and the performance are improved. On the other hand, they have studied the effects of the shared caches by only up to 4 cores. The number of cores in modern vector processors is increasing, and this trend is expected to continue. Hence, a vector processor with more vector cores should be studied for the future. Additionally, they do not consider the cache associativity.

Egawa *et al.* [3, 4] have studied the shared cache up to 16 cores using real applications in a multi-core vector processor and clarified that increasing the number of cores and capacity of shared cache improves the hit rate. They have also clarified the improvement of the performance efficiency and the reduction in power consumption by the shared cache. Besides, they confirmed that an 8MB shared cache configuration is the most efficient for a 16-core vector processor. However, their studies are only for the number of cores and the cache capacity, not for the cache associativity. Additionally, the gap between the computing capability and the memory bandwidth has further widened after their study has done.

In order for the vector processors to obtain the high sustained performance even with increasing the number of vector cores, it is necessary to consider the cache associativity, as discussed in Section 1.3. Therefore, this paper studies the shared cache configurations for vector architectures in terms of the cache associativity and the number of cores sharing one cache, assuming that the number of vector cores significantly increases in the future.



**Figure 6.** The difference of the 2-way set-associative and the 2-way skewed-associative

## 2. Skewed Multi-banked Cache for Many-core Vector Processors

Skewed-associativity [13] is an effective method to eliminate conflict misses without increasing the associativity. Hence, this paper proposes the skewed cache for the many-core vector processors in order for the shared cache to reduce the number of conflict misses on vector load/store data.

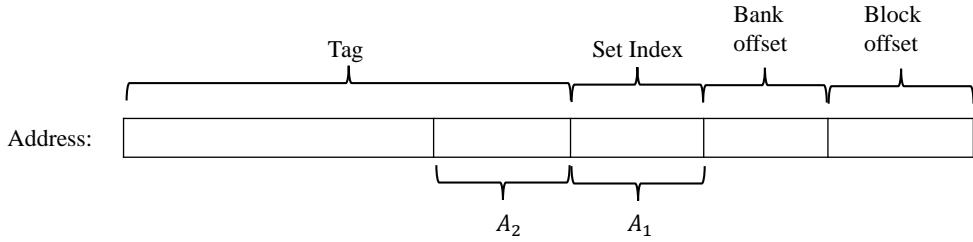
### 2.1. Skewed Cache

A skewed cache eliminates conflict misses by allocating a block to a set using the hashing function for each way. Figure 6 shows the difference between a 2-way set-associative cache and a 2-way skewed cache.

In Fig. 6(a), the set-associative cache allocates addresses D1, D2, and D3 to the same set. Thus, if the blocks of these addresses are stored in this order, the block of D1 inserted at first will be evicted by inserting the block of D3 due to the set conflict. In contrast, in Fig. 6(b), hashing function  $f_0$  for way 0 and hashing function  $f_1$  for way 1 individually generate different set indices so that these addresses are indexed to different sets. Since the block of D3 is stored to the different set where that of D1 is indexed to, it is possible to avoid evicting the block of D1 and avoid causing a conflict miss.

In order to obtain the high cache hit rate on the skewed cache, the hashing function and the replacement policy are essential. The hashing function intrinsically affects the ability of the skewed cache to avoid set conflicts. Thus, the hashing function should output non-biased values; otherwise, blocks are placed in the same set, causing conflict misses. Furthermore, it is desirable to easily create the various hashing functions based on a single rule so that they can produce different outputs for each way.

The replacement policy is also essential. One of the well-known problems with the skewed cache is that it is challenging to implement the Least Recently Used policy (LRU) on the skewed cache in the case of a higher associativity of three or more. This is because LRU generally determines an evicted block by using insertion order of blocks in the set. In the case of the set-associative cache, replacement candidates are always chosen from the same set, and an evicted block is selected from them. Thus, it is necessary to keep the insertion order within the sets. However, the skewed cache chooses replacement candidates from the different sets depending on the address of the block and the hashing function, so that the insertion order within the sets cannot determine whether the blocks recently used or not. If absolute timestamps are added to every block, LRU can be realized for the skewed cache. However, it requires an impractical hardware cost.



**Figure 7.** The bit field of hashing function of the given address

The remaining of this section discusses the implementations of the hashing function and the replacement policy that significantly affects the performance and implementation cost of the skewed cache.

## 2.2. Hashing Functions

### 2.2.1. XOR-based hashing function

An exclusive OR (XOR) based hashing function based on the XOR operations is used when the skewed-associativity is proposed [2, 13, 14]. The following equation calculates the set index.

$$index = \sigma^w(A_2) \oplus A_1 \bmod n_{set}, \quad (4)$$

in which  $\oplus$  represents the bitwise exclusive OR operator.  $A_1$  and  $A_2$  denote fields of an address in Fig. 7. The bit lengths of  $A_1$  and  $A_2$  are  $\log_2(n_{set})$ .  $\sigma^w$  represents a  $w$ -bit circular shift operation. Here,  $w$  generally represents the way ID for each way. For example, the hashing function for way 0 is  $A_2 \oplus A_1 \bmod n_{set}$ , and that for way 1 is  $\sigma^1(A_2) \oplus A_1 \bmod n_{set}$ .

The advantages of the XOR-based hashing function are simpleness for implementation, and this function satisfies two requirements for skewing. The blocks that may be mapped to the same set in a way are spread over other sets of the other ways. Moreover, two blocks with the same higher bits ( $A_2$  or tag field in Fig. 7) cannot be mapped to the same set by the function.

However, the XOR-based hashing function is known to suffer from specific stride patterns called *pathological behavior* [7]. For example, if  $n_{set}$  is 16,  $A_1$  is fixed, and  $A_2$  varies as a stride of 8, the XOR-based hashing function will generate the sequence of set indices 1,9,1,9,...,1,9. The same problem occurs even when  $w$  changes. Furthermore, if either  $A_1$  or  $A_2$  is 0 or 11..111, the outputs stay the same value despite any  $w$ . Thus, this paper examines another hashing function for the skewed cache.

### 2.2.2. Odd-multiplier displacement hashing function

This paper examines the Odd-Multiplier Displacement Hashing Function (oDisp) [7] as the alternative hashing function. the oDisp is known as a more uniform hashing function than the XOR-based hashing function. Thus, even in the case where access patterns contain strides of a specific length, the oDisp can be expected to reduce conflict misses further. The following equation expresses the set index

$$index = (o \times A_2 + A_1) \bmod n_{set}, \quad (5)$$

where  $o$  is an arbitrary odd number changed for each way.

```

Require: Candidates
Ensure: An evicted block
1: if counter > (the number of blocks / 4) then
2:   Reset RU of all blocks
3:   Set counter 0
4: else
5:   Increment counter
6: end if
7: if Cache hits then
8:   Set RU of the hit block to 1
9: else if Cache misses then
10:  g1 := Candidates.where(RU is 0)
11:  g2 := Candidates.where(RU is 1 and clean)
12:  g3 := Candidates.where(RU is 1 and dirty)
13:  if g1 is not empty then
14:    Return one randomly from g1
15:  else if g2 is not empty then
16:    Return one randomly from g2
17:  else if g3 is not empty then
18:    Return one randomly from g3
19:  end if
20: end if

```

---

**Figure 8.** The flow of NRUNRW policy

Another advantage of this hashing function is that the hardware cost is low. Equation (5) means that an arbitrary odd number multiplied by  $A_2$  is added to  $A_1$ , and the remainder is calculated by the number of sets. Multiplication of an arbitrary odd number can be realized by one logical shifter and two adders instead of a multiplier. Furthermore, it is easy to change the output of the hashing function for each way by selecting different odd numbers for each way. Therefore, the index of the oDisp can be simplified by the following equation

$$index = ((A_2 \ll w) + A_2 + A_1) \bmod n_{set}, \quad (6)$$

where  $w$  represents the way number and  $\ll$  represents a logical left-shift operation.

### 2.3. Replacement Policies

#### 2.3.1. Not recently used not recently written

There are some studies about a replacement policy for the skewed cache, Seznec *et al.* [15] have proposed the Not Recently Used Not Recently Written policy (NRUNRW) based on the Not Recently Used (NRU). Figure 8 shows the flow of NRUNRW. This policy requires one bit per cache block as a Recently Used bit (RU) and selects one of the non-latest cache blocks as an evicted block. If the policy cannot determine the evicted block depending on the RUs of candidate blocks, the policy selects the evicted block from clean blocks among the candidate blocks. Furthermore, all RUs in the cache is reset every interval when the number of requests to the cache reaches one-fourth of the number of the total cache blocks.

```

Require: Candidates
Ensure: An evicted block
1: if Cache hits then
2:   Set RRPV of the hit block to 0
3: else if Cache misses then
4:   while True do
5:     for c in Candidates do
6:       if c.RRPV is 3 then
7:         Return c
8:       end if
9:     end for
10:    Increment RRPVs of all candidates
11:   end while
12: end if

```

---

**Figure 9.** The flow of SRRIP policy

The advantage of NRUNRW is that it requires a small hardware cost of only one bit per cache block. However, Seznec [14] pointed out that the performance of NRUNRW did not reach that of LRU because of the randomness by resetting the RUs. Therefore, this paper also examines an alternative replacement policy for the skewed cache.

### 2.3.2. *Static Re-Reference Interval Prediction*

In order to obtain the hit rate equivalent to LRU at a realistic hardware cost, the proposed skewed cache adopts the Static Re-Reference Interval Prediction policy (SRRIP) [6] as a replacement policy.

SRRIP is implemented as an extension of NRU and can achieve hit rates comparable to LRU with a simple replacement algorithm and low hardware cost. Fig. 9 shows the flow of SRRIP. SRRIP prepares an  $m$ -bit Re-Reference Prediction Value (RRPV) per cache block and predicts when the block will be re-referenced. SRRIP takes advantage of the fact that recently hit blocks have higher reusabilities than newly-inserted blocks. Since SRRIP lets hit blocks be hardly evicted, newly-inserted blocks can be evicted earlier than the recently hit block. Therefore, only recently hit blocks tend to remain in the cache, and other blocks are replaced.

SRRIP can be used for the skewed cache because the information used when replacing blocks, i.e. an RRPV for each block, can predict the re-reference interval of a block independent from the RRPVs of the other blocks in the same set. Thus, an evicted block can be determined even if replacement candidate blocks come from different sets, as in the case of the skewed cache.

## 3. Evaluation

### 3.1. Experimental Environment

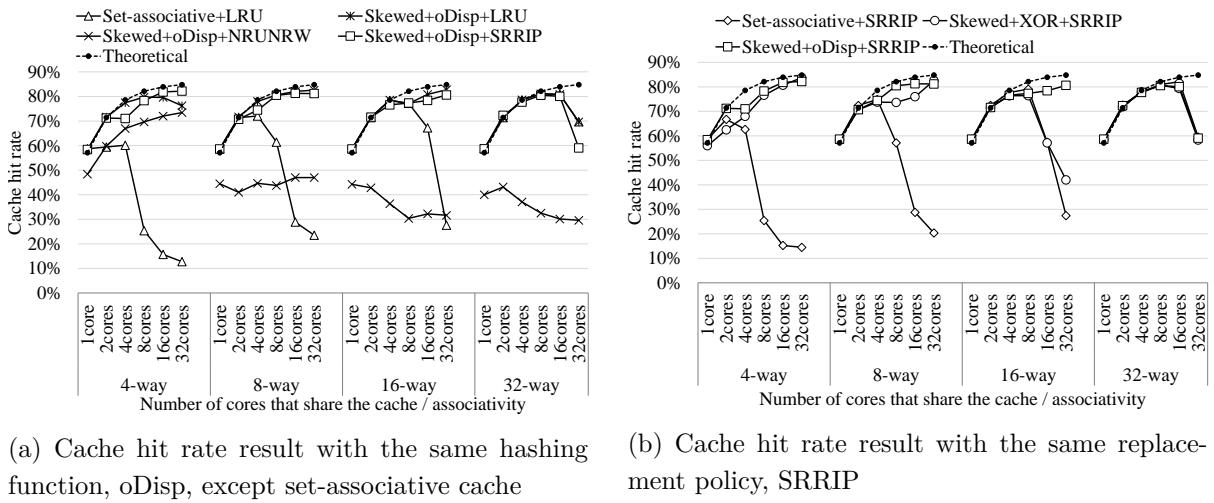
In order to confirm the effect of the skewed cache on the performance of many-core vector processors, we conduct experiments. The simulator of a many-core vector processor is developed based on the gem5 simulator [1], which is a general-purpose architecture simulator. The simulator uses an instruction trace data as an input, which is obtained by the vector supercomputer SX-ACE. Based on the trace data, it simulates the occupancy of hardware resources inside the processor and calculates the various performance metrics.

The simulation of the many-core vector processor is performed by implementing pseudo cores. The pseudo cores only issue requests to the memory system so that a simple implementation of pseudo cores enable to simulate many-core vector processors at high speed. In the stencil calculation targeted in this paper, it is assumed that parallelization is performed in the outermost loop of Fig. 3 discussed in Section 1.2.2. Therefore, the widths specified for the pseudo cores correspond to the amount of data for one z-iteration. The space of the calculation is set to 2048x2048x512.

Table 1 shows the system configurations of the many-core vector processor. The total number of cores is set to 32 based on the trend in the future many-vector core processor, and the configuration of the core is based on that of NEC SX-ACE [5]. In addition, this paper assumes that the Bytes/Flop (B/F) value of the system used for the evaluation is set to 0.125. This is because the trend in the gap between computing capability and memory performance has widened, and the B/F value is diminishing. In fact, the B/F values of SX-ACE and SX-Aurora TSUBASA are 1.0 and 0.5, respectively. The newer generation vector processors ought to have lower B/F values.

**Table 1.** Configurations of the simulation

Base architecture	NEC SX-ACE
Total number of core	32
Number of cores sharing a same cache	1, 2, 4, 8, 16, 32
Main memory bandwidth	256GB/s
Total cache size (size per bank)	32MB (128KB)
Associativity	4, 8, 16, 32
Cache block size	128Bytes
MSHR (Target) [9]	8 (8)
System B/F	0.125 B/F


**Figure 10.** Cache hit rate results

In the evaluation, several parameters affect the performance of the multi-banked cache memory. The associativity is set to 4, 8, 16, and 32, and the number of cores sharing one cache is set to 1, 2, 4, 8, 16, and 32. The total capacity and the total number of banks are set to 32MB and 256 banks, respectively. The total capacity is fixed during the evaluation to focus on only the effect of the shared cache, and the total number of banks is also fixed to keep the bandwidth the same in all the configurations.

In addition to the proposed skewed cache, we evaluate the conventional set-associative cache to compare the results. On the proposed cache, NRUNRW, LRU, and SRRIP policies are used as replacement policies. This paper sets two bits to the RRPV for SRRIP in the evaluation, which can achieve the performance comparable to LRU [6]. Since the implementation of LRU in the realistic hardware cost for the skewed cache is difficult, LRU is implemented as the insertion order of blocks is judged by the absolute timestamps given to blocks.

### 3.2. Evaluation Results and Discussion

#### 3.2.1. Cache hit rate

Figure 10 show the cache hit rates of the conventional cache and the skewed cache with various replacement policies and hashing functions. In the figures, the vertical axis represents the cache

hit rate, and the horizontal axis represents the number of cores sharing a cache and associativity of the cache.

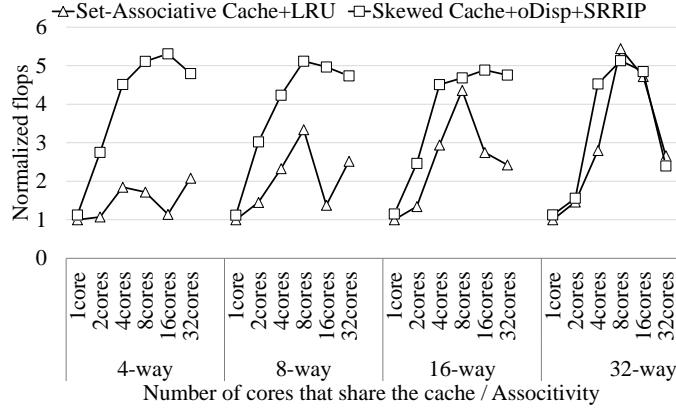
Figure 10(a) shows the cache hit rate with various replacement policies. In order to discuss the difference among the replacement policies, we use the odd-multiplier displacement hashing as the hashing function. The hit rates of the proposed skewed cache with SRRIP are very close to the theoretical hit rates for each configuration. SRRIP follows theoretical values only when the associativity is 4-way or 8-way, although the hit rates of the high associativities are slightly lower than those of LRU. Compared to the set-associative cache when the cache is 4-way and 8-way, the number of hits is improved by a maximum 70% and 60%, respectively. In the case where the 32 cores share a 16-way cache, the number of hits increases by about 50%.

In contrast, the cache hit rate drastically decrease with the low associativity in the set-associative cache. Particularly, when the associativity is 4-way, 8-way or even 16-way, and the number of cores sharing a cache is large, the gap between set-associative cache and theoretical hit rates becomes apparent. This is because the stencil calculation consists of relatively regular memory access patterns. In the evaluation, the stencil calculation is parallelized, as discussed in Section 1.2.2. Every core issues memory access requests of the z-iteration assigned to each core. However, a difference in the requests can be distinguished mainly by the tag field of their addresses. Thus, these requests tend to be indexed to the same set in the set-associative cache, which results in conflict misses. On the other hand, the skewed cache could avoid conflict misses since it uses a broader range of an address field including a part of the tag field so that the skewed cache can distinguish the difference of the z-iteration on the address to determine the set.

As a replacement policy of the skewed cache, the hit rate of LRU achieves almost equal to the theoretical value. When LRU is applied, the skewed cache can maintain a higher hit rate than the set-associative cache or a high hit rate equal to the skewed cache with SRRIP. This is because the skewed cache with LRU is based on the timestamp so that blocks can be ideally replaced. Although its implementation requires impractical hardware costs, it is possible to eliminate more conflict misses than SRRIP regardless of the associativity. Note that NRUNRW has a low hit rate in all cases. This is because all the RUs are reset at each fixed access interval so that an evicted block can be randomly selected every interval. It causes that the necessary blocks are suffering from random eviction.

When the number of cores sharing one cache is 16 cores or more, and the associativity is 32-way, the cache hit rate decreases on both SRRIP and LRU. Two reasons can be considered. The first reason is that the number of sets per way decreases if the associativity is too high. Therefore, the possibility to select the same set many times becomes high, resulting in increasing the number of conflict misses. The second reason is due to the implementation matter of the oDisp hashing functions. In the skewed cache, the oDisp hashing function is implemented, as shown in Equation (6). When the number of ways is larger than the number of bits of  $A_2$ , the shifted  $A_2$  does not affect the calculation of the index on the larger way numbers than the bitfield length of  $A_2$ . Therefore, these cache hit rates could become lower than those of the theoretical value. Note that the miss rates of SRRIP are slightly lower than those of LRU. When the associativity is very high, SRRIP meets many replacement candidates with the same RRPVs at once and becomes closer to a random replacement, which causes the increase in the number of misses.

Figure 10(b) shows the hit ratio of the skewed cache where a hashing function is changed. In order to discuss the difference among the hashing functions, we use SRRIP for the replacement policy of the skewed cache and the set-associative cache. From Fig. 10(b), it is observed that both



**Figure 11.** Performance comparison of the conventional set-associative cache and the proposed skewed cache

hashing functions can obtain a high cache hit rate across almost all cases. However, in the case where the associativity and the number of cores sharing one cache are extremely large, the XOR-based hashing function decreases the cache hit rate. Two reasons can be considered. The first reason is that the skewed cache with the XOR-based hashing function sometimes suffers from the specific access pattern called *pathological behavior*, as mentioned in Section 2.2.1. It is considered that, because the memory addresses requested from the vector cores coincidentally meet these kinds of specific stride pattern, the cache hit rate degradation is observed, specifically in the case of 16-way cache shared by 16 cores and 32 cores. The second reason is due to the implementation matter of the XOR-based hashing functions. When the number of ways is larger than the number of bits of  $A_2$  on the XOR-based hashing function, the bit rotation is circulated, and the same index is calculated again on the larger way numbers. This leads to the cache hit rate degradation. In contrast, the oDisp can take care of this kind of access patterns. Therefore, it can obtain a stable high cache hit rate by the oDisp. Note that the set-associative cache with SRRIP suffers from lowering the cache hit rate due to the same reason of the set-associative cache with LRU in Fig. 10(a).

### 3.2.2. Performance

Figure 11 shows the performance comparison of the set-associative cache with the proposed skewed cache. The set-associative cache is based on LRU, and the skewed cache with oDisp & SRRIP is the proposed cache that can achieve a high hit rate at a reasonable implementation cost. In Fig. 11, the vertical axis represents the performance normalized by the case where each core privately owns a set-associative cache. The horizontal axis shows the number of cores sharing a cache and the associativity. Figure 11 shows that the proposed cache outperforms the set-associative cache when the associativity is low. Notably, in the case where 16 cores share a 4-way cache, the skewed cache obtains a six times higher performance than the set-associative cache. On the other hand, if there is sufficient associativity, there are no significant differences in performances between the skewed cache and the set-associative cache. This is because, if there is sufficient associativity, fewer conflict misses occur in both cases.

In Fig. 11, there are typical cases where one cache shared by 16 cores, and its performance is higher than that of the case shared by 32 cores, although the cache hit rate in 32 cores is higher than that in 16 cores. It is due to cache bank conflicts. Since the cache configuration of the many-core vector processor assumes a multi-banked configuration in this paper, memory access requests

go to each cache bank according to the memory address. Therefore, when many cores share one cache, the possibility of accessing the same cache bank is increasing. Additionally, in this cache configuration, the write buffer size and MSHR per bank are only 16 and 8, respectively. Therefore, in the case where 32 cores share a cache, their requests are concentrated on some bank due to bank conflicts, and the shortages of the write buffer or MSHR cause the performance degradation.

Overall, the skewed cache can realize the better performance than the conventional set-associative cache, especially when the associativity is low. This is because the skewed cache can successfully eliminate conflict misses. It is clarified that SRRIP can bring almost ideal cache hit rate to the skewed cache at reasonable implementation costs. Moreover, the oDisp hashing function avoids performance degradation from conflict misses.

## Conclusions

This paper proposes a skewed multi-banked cache for many-core vector processors. The skewed cache can prevent simultaneously requested blocks from using the same cache set. The data block is stored for a cache set by using a hashed value of the block address. This paper discusses how the most important two features of the skewed cache should be implemented: the hashing functions and the replacement policies. This paper evaluates three replacement policies, LRU, NRUNRW, and SRRIP, and two hashing functions, XOR-based and oDisp for the skewed cache.

The evaluation results show that the skewed cache with SRRIP and oDisp can increase the number of hits by up to 70% and marks the closest results to those of the theoretical upper bound of the hit rate of the shared cache. From the individual evaluations of hashing functions and replacement policies, SRRIP can obtain the highest cache hit rate at low hardware cost, and oDisp can solve the problem of the XOR-based hashing function. The evaluation results also show that the skewed cache can realize a six times higher performance than the conventional set-associative cache on the stencil calculation. As future work, the skewed cache should be evaluated with more various real applications developed for modern vector processors.

## Acknowledgements

This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program, entitled "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications" and Grants-in-Aid for Early-Career Scientists No. 19K20232. The experimental results in this research were partially obtained by supercomputing resources at Cyberscience Center, Tohoku University.

*This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.*

## References

1. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., et al.: The Gem5 Simulator. SIGARCH Comput. Archit. News 39(2), 1–7 (2011), DOI: 10.1145/2024716.2024718
2. Bodin, F., Seznec, A.: Skewed associativity improves program performance and enhances predictability. IEEE Transactions on Computers 46(5), 530–544 (1997), DOI: 10.1109/12.589219

3. Egawa, R., Funaya, Y., Nagaoka, R., Endo, Y., Musa, A., Takizawa, H., Kobayashi, H.: Effects of 3-D stacked vector cache on energy consumption. In: 2011 IEEE Int. 3D Systems Integration Conf. (3DIC), 2011 IEEE Int. pp. 1–6 (2012), DOI: 10.1109/3DIC.2012.6263026
4. Egawa, R., Funaya, Y., Nagaoka, R., Musa, A., Takizawa, H., Kobayashi, H.: Design and early evaluation of a 3-D die stacked chip multi-vector processor. In: 2010 IEEE International 3D Systems Integration Conference (3DIC). pp. 1–8 (2010), DOI: 10.1109/3DIC.2010.5751448
5. Egawa, R., Komatsu, K., Momose, S., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Potential of a Modern Vector Supercomputer for Practical Applications: Performance Evaluation of SX-ACE. *J. Supercomput.* 73(9), 3948–3976 (2017), DOI: 10.1007/s11227-017-1993-y
6. Jaleel, A., Theobald, K.B., Steely, Jr., S.C., Emer, J.: High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38(3), 60–71 (2010), DOI: 10.1145/1816038.1815971
7. Kharbutli, M., Solihin, Y., Lee, J.: Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. *IEEE Trans. Comput.* 54(5), 573–586 (2005), DOI: 10.1109/TC.2005.79
8. Komatsu, K., Momose, S., Isobe, Y., Watanabe, O., Musa, A., Yokokawa, M., Aoyama, T., Sato, M., Kobayashi, H.: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 685–696 (2018), DOI: 10.1109/SC.2018.00057
9. Kroft, D.: Lockup-free Instruction Fetch/Prefetch Cache Organization. In: Proceedings of the 8th Annual Symposium on Computer Architecture. pp. 81–87. ISCA '81, IEEE Computer Society Press, Los Alamitos, CA, USA (1981), <http://dl.acm.org/citation.cfm?id=800052.801868>
10. Musa, A., Sato, Y., Soga, T., Okabe, K., Egawa, R., Takizawa, H., Kobayashi, H.: A Shared Cache for a Chip Multi Vector Processor. In: Proceedings of the 9th Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture. pp. 24–29. MEDEA '08, ACM, New York, NY, USA (2008), DOI: 10.1145/1509084.1509088
11. Qureshi, M.K., Thompson, D., Patt, Y.N.: The V-Way cache: demand-based associativity via global replacement. In: 32nd International Symposium on Computer Architecture (ISCA'05). pp. 544–555 (2005), DOI: 10.1109/ISCA.2005.52
12. Sanchez, D., Kozyrakis, C.: The ZCache: Decoupling Ways and Associativity. In: 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 187–198 (2010), DOI: 10.1109/MICRO.2010.20
13. Seznec, A.: A Case for Two-way Skewed-associative Caches. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 169–178. ISCA '93, ACM, New York, NY, USA (1993), DOI: 10.1145/165123.165152
14. Seznec, A.: A New Case for Skewed-Associativity. Research Report RR-3208, INRIA (1997), <https://hal.inria.fr/inria-00073481>
15. Seznec, A., Bodin, F.: Skewed-associative caches. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE '93 Parallel Architectures and Languages Europe. pp. 305–316. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)